DISS. ETH NO. 20220

MODELING, COMPILING, AND EFFICIENTLY EXECUTING BUSINESS PROCESSES ON RESOURCE-CONSTRAINED WIRELESS SENSOR NETWORKS

A dissertation submitted to

ETH ZURICH

for the degree of
Doctor of Sciences

presented by
ALEXANDRU MIRCEA CARACAŞ
M.Sc. in Computer Science, International University in Germany

born 14.12.1981

citizen of Romania

accepted on the recommendation of

Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Gustavo Alonso, co-examiner
Dr. Thorsten Kramp, co-examiner
Dr. Jan Beutel, co-examiner

2012

Abstract

A process describes a transformation which advances a system from an initial starting point to a final state. The processes that govern the production of goods and delivery of services in today's economy are called business processes. These processes add economic value to both the company and the respective customers. Such a business process specifies the sequence of activities, events, and interactions between humans, information technology systems and the physical environment in order to fulfill a specific business goal. Over the past two decades, several standard languages have evolved which allow business processes to be described using graphical models.

In parallel to these developments, the performance of information technology systems increased following Moore's law. Embedded devices are continuously being miniaturized to the point where they become ubiquitous. Consequently, the technology offered by *wireless sensor networks* (WSN) allows business processes to be embedded even more deeply into the physical environment. In this way, business processes can better monitor and interact in real time with the entities they control, thus decreasing operational costs and improving responsiveness. To reduce investment costs and energy consumption, WSN are built using the low-end spectrum of embedded devices. The limited set of resources of such devices poses a significant challenge for the integration of WSN into business processes. As a consequence, classical implementations of business processes on embedded WSN are specific, one-of-a-kind solutions, an approach that is inflexible and expensive with respect to changing business requirements.

To align the implementation of a WSN application with the respective business processes, this thesis proposes a model-driven approach. Business domain experts can specify process models which describe the business-relevant behavior of WSN applications. These process models can then be further refined using different abstraction levels. The building blocks for models on the lowest abstraction level provide basic functionality and drivers. These artifacts are implemented using traditional software development methods. Using code generation techniques, the high-level business process models and the software artifacts are compiled into executable WSN applications. Thus, organizations can use WSN technology to adapt to a dynamic competition situation faster and can meet the demands of a diverse customer base more quickly. The business-relevant behavior of remote sensors and actuators becomes visible in models and is fully aligned with the corresponding implementation. Furthermore, graphical models foster communication among the different business participants and technology implementers using the same lingua franca.

The proposed model-driven approach for WSN applications requires a set of tools which includes a model editor, a compiler, and a simulation and testing environment. To fully benefit from the modeling abstraction, the tools must allow the testing of the behavior of a business process prior to deployment. To minimize energy consumption and memory usage, the compiler must generate efficient code which takes advantage of specific power management features of the underlying run-time platform. The proposed modeling approach and run-time optimizations are applicable to other graphical modeling languages and to different WSN execution platforms. As a concrete example, this thesis focuses on the *business process model and notation* (BPMN) standard as the modeling language. To address a large business audience and leverage the extensive set

of tools, the proposed methodology allows WSN applications to be described without changing either the syntax or the execution semantics of the modeling language.

The evaluation of the proposed methodology focuses on expressability and efficiency. These aspects are assessed using our implementation of the required tools. To show the expressability of the approach, we selected a set of archetype WSN applications, which we modeled using our tools. We then analyzed the efficiency of the automatically generated application in terms of energy and memory consumption when compared with the corresponding manually written implementation. The benefits of our approach, such as the graphical overview over processes, a business-aligned implementation, and the flexibility to adapt to new business requirements, come at the price of 10% more RAM and 44% more flash space consumption on average. However, the evaluation results also show that the generated code can be executed on resource-constrained sensor nodes consuming only 1% more energy than the manually written equivalents.

Zusammenfassung

Ein Prozess beschreibt eine Transformation, die ein System von einem Ausgangszustand in einen Endzustand überführt. Die Prozesse, die die Güterproduktion und Dienstleistungen in der heutigen Wirtschaft regeln, sind sogenannte Geschäftsprozesse (Business Processes). Diese steigern den wirtschaftlichen Wert des Unternehmens und für die jeweiligen Kunden. Ein Geschäftsprozess beschreibt wie ein bestimmtes Geschäftsziel zu erfüllen ist, nämlich durch Abfolgen von Aktivitäten, Ereignissen sowie Interaktionen zwischen Menschen, Informationstechnik-Systemen und der physischen Umwelt. Mehrere Standard-Sprachen, die in den letzten zwei Jahrzehnten entwickelt wurden, erlauben die Beschreibung von Geschäftsprozessen mittels grafischer Modelle.

Parallel zu diesen Entwicklungen stieg die Leistung der Informationstechnik gemäss dem Moorschen Gesetz. Eingebettete Geräte wurden kontinuierlich miniaturisiert bis zu dem Punkt, an dem sie allgegenwärtig geworden sind. Folglich ermöglichen *drahtlose Sensornetze* (WSN), Geschäftsprozesse noch tiefer in die physische Umwelt einzubetten. Auf diese Weise können Geschäftsprozesse besser und in Echtzeit die von ihnen kontrollierten Objekte überwachen und mit ihnen interagieren. Dadurch werden die Betriebskosten gesenkt und die Reaktionsfähigkeit verbessert. Um Investitionskosten und Energieverbrauch zu reduzieren, bestehen WSN typischerweise aus Systemen, die bezüglich der benötigten Ressourcen am untere Ende der Skala zu finden sind, sogenannte "low-end" Systeme. Die beschränkte Ressourcen solcher Systeme stellen eine erhebliche Herausforderung für die Integration von Sensornetzen in Geschäftsprozesse dar. Folglich sind klassische Implementierungen von Geschäftsprozessen auf WSN sehr spezifische Lösungen, ein Ansatz der teuer und inflexibel ist, speziell wenn wechselnde Unternehmensanforderungen in Betracht gezogen werden.

Um die Implementierung einer WSN-Anwendung mit den jeweiligen Geschäftsprozessen in Übereinstimmung zu bringen, schlägt die vorliegende Arbeit einen modellbasierten Ansatz vor. Domänen-Experten können Prozessmodelle erstellen, die geschäftsrelevante Verhaltensweisen von WSN-Anwendungen beschreiben. Diese Prozessmodelle können auf unterschiedlichen Abstraktionsebenen weiter verfeinert werden. Die Bausteine für die Modelle bieten auf der untersten Abstraktionsebene grundlegende Funktionen und Treiber an, die mit traditionellen Methoden der Softwareentwicklung implementiert werden. Mithilfe Code-generierender Methoden werden Geschäftsprozessmodelle und Software-Artefakte zu einer ausführbaren WSN-Anwendung kompiliert. Auf diese Weise können Organisationen WSN-Technologien verwenden, um sich schneller an ein dynamisches Wettbewerbsumfeld anzupassen und sie können damit schneller den Anforderungen eines vielfältigen Kundenstammes gerecht werden. Das geschäftsrelevante Verhalten von Sensoren und Aktoren wird dadurch in Modellen sichtbar und stimmt damit vollständig mit der entsprechenden Umsetzung überein. Darüber hinaus fördern grafische Modelle die Kommunikation zwischen den verschiedenen Akteuren in unterschiedlichen Geschäftsbereichen und den technischen Entwicklern.

Der vorgeschlagene Modellbasierte Ansatz für WSN-Anwendungen erfordert eine Reihe von Werkzeugen. Konkret einen Modell-Editor, einen Compiler und eine Simulations- und Testumgebung. Um vollständig von der Modellierungsabstraktion zu profitieren, sollen die Werkzeuge es ermöglichen, das Verhalten

eines Geschäftsprozesses vor dessen Einsatz zu testen. Zur Minimierung des Energieverbrauchs und der Speichernutzung muss der Compiler effizienten Code generieren, der wiederum die spezielle Energieverwaltungs-Funktionalität der zugrunde liegenden Laufzeit-Plattform nutzt. Die vorgeschlagene Modellierung und die Laufzeit-Optimierungen sind für verschiedene grafische Modellierungs-Sprachen und für verschiedene WSN-Ausführungsplattformen anwendbar. Als konkretes Beispiel konzentriert sich diese Arbeit auf den *Geschäftsprozessmodell und Notation* (Business Process Model and Notation — BPMN) Standard als Modellierungssprache. Um ein grosses Geschäftspublikum, das bereits die umfangreiche Sammlung von Werkzeugen kennt, anzusprechen, erlaubt die vorgeschlagene Methodik WSN-Anwendungen zu beschreiben, ohne dass an der Syntax oder an der Ausführungs-Semantik der Modellierungssprache Änderungen vorgenommen werden müssten.

Die Auswertung der vorgeschlagenen Methodik konzentriert sich auf Ausdrucksfähigkeit und Effizienz. Diese Aspekte werden anhand unserer Implementierung der erforderlichen Werkzeuge evaluiert. Um die Ausdruckfähigkeit des Ansatzes zu zeigen, benutzen wir eine Reihe von typischen Anwendungen für WSN, die wir dann mit unseren Werkzeugen modellieren. Danach analysieren wir die Effizienz der automatisch generierten Anwendungen in Bezug auf die Energie- und Speicher-Ressourcen im Vergleich mit dem jeweiligen entsprechenden manuell implementierten Äquivalent. Die Vorteile unseres Ansatzes, wie die grafische Übersicht über die Prozesse, eine mit dem wirklichen Geschäftsablauf übereinstimmende Implementierung und die Flexibilität, schnell auf neue Geschäftsanforderungen umzustellen, erfordern jedoch einen durchschnittlich 10% grösseren RAM-Bedarf und 44% mehr Flash-Speicherplatzbedarf. Hingegen zeigen die Ergebnisse der Evaluation auch, dass die Ausführung des generierten Quellcodes auf Sensorknoten mit beschränkten Ressourcen nur 1% mehr Energie verbraucht als Anwendungen, die manuell implementiert wurden.

Preface

The process of writing a thesis is much like a long-distance marathon. When starting the race, the relief map of the terrain along the parcours is unknown. Thus, along the way one experiences both uplifting parts and downward parts. Luckily, one is never traveling alone on this endurance journey. Peers may cross your run on some parts of the path, and friends and family offer refreshments along the way. It is thus not surprising that such a race affects not only you as the runner but, to a great extent, also the people around you. In the next lines, I would like to thank the people along my marathon race for their energetic support, which allowed me to concentrate on reaching the finish line.

First, I would like to thank my PhD supervisor, Prof. Friedeman Mattern, and my supervisor at IBM, Thorsten Kramp, for their guidance, continuous support, critical and constructive reviews, and ensuring that my research stays on track. Equally, I would like to thank my co-examiners Prof. Gustavo Alonso and Jan Beutel for their time and challenging discussions. In all respects, I am looking forward to future collaborations.

Second, I would like to thank all my ETH research peers and colleagues. In particular, I thank Alexander Bernauer for the challenging and interesting discussions and fruitful collaborations. Equally, I would like to thank all my colleagues from the IBM Mote Runner team and the collaborators from IBM Research. In particular, I thank Thomas Eirich and Marcus Oestreicher for their experienced insights and improvement suggestions, helping with long and often time-consuming debugging sessions both on the hardware and in the simulation, and providing constructive and critical reviews. On the same line of thought, I also thank Yvonne-Anne Pignolet, Clemens Lombriser, Urs Hunkeler, Rolf Adelsberger, and Hong Linh Truong.

During the marathon race, a transformation occurs. The runner gains new experiences and learns how to address problems from different perspectives. This continuous learning is possible through interactions and discussions with friends, peers and colleagues. I would like to thank them all by selecting some activities they can relate to: designing special-purpose hardware and manually soldering the tiniest of components, plotting graphs and analyzing data produced by oscilloscopes and other obscure lab equipment using MatLAB functions, passionate discussions on security and privacy in social networks, motivating me to run my first Olympic-distance triathlon, numerous mind-clearing sport activities, some times over the lunch break (running and swimming) and other times outside of the lab premises (basketball, climbing, skiing), sharing their own PhD experiences, and offering many practical tips ranging from bank accounts and insurances down to sports equipment and electronic gadgets.

Finally, I am grateful for all the support and encouragements from my family and the care and strength of my wife, who shared many of the worries and successes along the marathon. Now this long-distance race is over and even greater challenges lie ahead. It is my hope that I will be able to return the same help and support.

Contents

1 Introduction **1**
- 1.1 Wireless Sensor Networks . . . 2
- 1.2 Business Process Management . . . 3
- 1.3 Business Requirements . . . 5
- 1.4 Classical Process Implementation . . . 6
- 1.5 Problem Set . . . 7
- 1.6 Thesis Goals . . . 8
- 1.7 Approach . . . 9
- 1.8 Challenges . . . 10
- 1.9 Methodology . . . 10

2 Model-Driven Development **11**
- 2.1 Business Process Modeling . . . 12
  - 2.1.1 Business Process Model and Notation (BPMN) . . . 12
  - 2.1.2 Event-driven Process Chain (EPC) . . . 14
- 2.2 Process Execution . . . 15
- 2.3 System Development . . . 16
  - 2.3.1 Specification and Description Language (SDL) . . . 17
  - 2.3.2 Universal Modeling Language (UML) . . . 18
  - 2.3.3 Data Flow Models . . . 20
  - 2.3.4 Domain Specific Languages (DSL) for WSN . . . 22
  - 2.3.5 Discussion . . . 23
- 2.4 Aligning WSN and Business Processes . . . 24
  - 2.4.1 A Gap Remains . . . 25
  - 2.4.2 Our Design Decisions . . . 26

3 Modeling WSN Applications using BPMN **29**
- 3.1 Background BPMN Terms and Concepts . . . 29
  - 3.1.1 Entities . . . 32
  - 3.1.2 Activities . . . 32
  - 3.1.3 Gateways . . . 33
  - 3.1.4 Data . . . 33
  - 3.1.5 Communication . . . 34
  - 3.1.6 Events . . . 34
- 3.2 Mapping WSN to BPMN . . . 35
  - 3.2.1 Reactive-Behavior . . . 36
  - 3.2.2 Heterogeneous . . . 41
  - 3.2.3 Communication . . . 43

3.2.4 Real Time . . . 49
3.2.5 Synchronization and Data Races . . . 50
3.3 Example Use Case . . . 52
3.3.1 The `Headquarters` Model . . . 53
3.3.2 The `Parcel` Model . . . 54
3.3.3 The `monitor` Model . . . 55

4 From BPMN Models To WSN Executables 57
4.1 Execution Semantics . . . 57
4.1.1 Activation and Logic . . . 58
4.1.2 Parallel Instances . . . 59
4.2 Compiler Algorithm . . . 61
4.2.1 State of the Art . . . 62
4.2.2 Our Contribution . . . 63
4.3 Compilation Patterns . . . 66
4.3.1 Data Flow . . . 67
4.3.2 Communication Flow . . . 68
4.3.3 Control Flow . . . 69
4.3.4 Events . . . 75
4.4 A Slim BPMN Run-Time . . . 81
4.5 Parallelism Detection Algorithm . . . 87

5 Integrated Development Tools 91
5.1 Model Editor . . . 91
5.1.1 Existing Editors . . . 92
5.1.2 Our Extensions . . . 93
5.1.3 Importing a Platform API . . . 93
5.2 Model Compiler . . . 97
5.3 Model Debugger . . . 99
5.4 The WSN Development Environment . . . 101
5.4.1 Mote Runner . . . 101
5.4.2 Simulation Environment . . . 104
5.4.3 Platform Debugger . . . 105
5.4.4 Integration and Visualization . . . 106
5.4.5 Deployment and Maintenance . . . 109
5.5 Process-Driven Development for WSN . . . 110
5.5.1 Modeling . . . 110
5.5.2 Implementation . . . 111
5.5.3 Testing . . . 112
5.5.4 Execution . . . 113
5.6 Implementation Considerations . . . 113
5.6.1 Web-based Editor . . . 113
5.6.2 Debugging View . . . 114
5.6.3 Platform API Toolbox . . . 115
5.6.4 Compiler Hints . . . 115

6 Optimizations **117**
6.1 Resource Usage . . . 117
6.1.1 Energy Consumption . . . 118
6.1.2 Memory Consumption . . . 120
6.2 Run-Time Power Management . . . 122
6.2.1 Optimized Sleeping Strategy . . . 122
6.2.2 Micro-Managing Communication . . . 125
6.3 Generated Code Optimizations . . . 129
6.3.1 Reusing Buffers . . . 130
6.3.2 Reducing The Number of Timers . . . 131
6.3.3 Preferring Static Execution . . . 132
6.3.4 Removing Dead Code . . . 133
6.3.5 Further Considerations . . . 133

7 Evaluation **135**
7.1 Expressiveness . . . 136
7.1.1 Extension Equivalence . . . 137
7.1.2 Application Domain . . . 138
7.1.3 A Note on Modeling . . . 140
7.1.4 Limitations . . . 141
7.2 Efficiency . . . 142
7.2.1 Run-time . . . 142
7.2.2 Generation . . . 146
7.2.3 Break-even (Dynamic vs. Static) . . . 150
7.3 A Note on Usability . . . 152

8 Conclusion and Future Directions **155**
8.1 Summary of Results . . . 155
8.2 Conclusion . . . 156
8.3 Outlook . . . 157

9 Annex **159**
9.1 Example Models . . . 159
9.2 Input Format . . . 161
9.3 Generated Source Code . . . 162

Chapter 1

Introduction

A *process* is synonymous with various transformations that advance a system from an initial state to a final state. The etymological roots of the noun process stem from the participle *processus* of the Latin verb procedere, which means to advance. Starting from this definition, the word process has slightly different meanings in various scientific domains, such as mathematics, medicine, biology, chemistry, physics, computer science, and business. For example, control theory is an established discipline which aims to automatically regulate dynamic systems using so-called control processes. In statistics, a well-established method for describing random phenomena are stochastic processes. In today's computer terminology, the word process is commonly used to describe an instance of a running program which is managed by an operating system.

process

The focus of this thesis is on *business processes* according to the following definition. In business terms [3], a process specifies the sequence of activities which guide work as well as the interactions between humans, information technology (IT) systems, and the environment to realize a business goal which adds economic value for the customer. Such processes are the driving force behind the operation of companies, the production of goods, and the execution and delivery of services in today's dynamic economy.

business processes

Historically, in 1776, the father of modern economy, Adam Smith, was among the first to describe a business process [1] for the manufacturing of a pin using a set of highly simplified and specialized operations:

> "One man draws out the wire, another straights it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving the head; to make the head requires two or three distinct operations; to put it on is a particular business, to whiten the pins is another; it is even a trade by itself to put them into the paper; and the important business of making a pin is, in this manner, divided into about eighteen distinct operations, which, in some manufactories, are all performed by distinct hands, though in others the same man will sometime perform two or three of them."

To remain competitive, companies constantly strive to improve performance, decrease operational costs, and provide better customer value by automating and appropriately managing their business processes. The rapid technological advancements during the industrial revolution in the late 19th century played a

key role in optimizing the individual operations in a process even further using machines for automation. The scientific analysis of modern production workflows, which aims to improve economic efficiency and labor productivity, can be traced back to the early 1900's with Taylor's management theories [163]. Control theory established the mathematical foundations for controlling dynamic systems [88] as far back as the 1930's starting with Black's negative feedback amplifiers and Nyquist's stability criterion followed by Bellman's dynamic programming. These theories later found application in control processes for industrial machinery and robots which relieved humans from repetitive and arduous activities in production lines. These methods focus mainly on optimizing the individual operations of a production line workflow. In the mid 1980's, improving the production line automation was perfected by Motorola's Six Sigma process management [19] based on statistical measures, which focused on eliminating defects.

In parallel, computing technologies evolved and became accessible to businesses in the second part of the 20th century. Consequently, processes with a well-defined sequence of activities and interactions which are executed frequently and in which a large amount of information processing is required are the prime candidates for automation using IT. Automation of such processes in an office dates back to the early 1980's when so-called workflow management systems were introduced to control information processing [38, 96].

In the beginning of the 1990's, the notion of business process engineering and redesign was introduced by Davenport [30] and Hammer [64]. Instead of using more technology to improve the efficiency of activities which do not add value, the goal of re-engineering is to use IT to remove them from a business process. Following this method, in the late 1990's, a milestone for business processes re-engineering (BPR) was set by the reusable business process models of SAP R/3 blueprints [29]. The BPR approach goes beyond the optimization of individual process steps through automation and allows a complete restructuring of a business by creating new processes which cut across organizational boundaries.

In today's IT-enabled environment, a business process is more than a simple description of a highly optimized sequential chain of activities as in Smith's pin factory example. A business process is a comprehensive description which includes details about all the entities involved in the process with all their interactions, all the data on which the process operates, all events occurring during the process, including possible exceptions from normal behavior, as well as organizational and resource information. Consequently, IT plays a key role in orchestrating the complex system interactions while monitoring and managing the overall business process.

1.1 Wireless Sensor Networks

Alongside the developments in the business processes space, IT systems evolved at a rapid pace. Following Moore's law, embedded devices are continuously being miniaturized to the point where they become ubiquitous. Nowadays, information processing and automation extend even further into the fabric of the environment, beyond production lines and offices, with wireless sensor networks (WSN) acting as remote eyes and hands for business processes.

WSN are increasingly being used by business processes to monitor and interact in real time with the entities they control. Examples include safety processes for storing hazardous materials [161], WSN securing the shipping and handling of containers to comply with government regulations [151], parcel delivery of perishable goods [78], systems to help automatic hospital processes [85], patient care [94], precision farming [10], as well as irrigation [105], and water management [128].

Such scenarios often describe remote and harsh sensing environments in which physical access to the sensor nodes and their batteries is not an option. Thus, sensor nodes have to run on their initial set of batteries or on harvested energy, often not spread evenly in the network and in many cases not satisfying the demand. For a long lifetime, energy must be minutely managed in all phases of operation. In addition, WSN applications are characterized by their distributed nature, event-based reactive behavior, unreliable wireless links, noisy sensors readings, real time requirements, and usage of heterogeneous platforms. A detailed description of WSN and their challenges is given in [185].

To enable better integration into the environment, reduce power consumption, and decrease production and deployment costs [142], WSN rely on the lower-end spectrum of embedded devices, such as 8-bit MCUs with 128 kB of non-volatile memory and 8 kB of volatile memory. Because of their relatively small size, the devices in a sensor network are called *motes*.[1] These constraints represent significant challenges when implementing business process using WSN.

1.2 Business Process Management

Automation focuses on improving the individual steps in a process as well as the entire process execution. Process automation is only one part in the entire business process management (BPM) cycle. The BPM methodology aims at improving businesses by redesigning them with the aid of IT and is the modern evolution of the previously introduced BPR techniques.

Figure 1.1 shows the main parts of the BPM cycle, which typically starts by describing or modeling a process. Before the process is implemented, its model can be simulated [79, 169, 184] using statistical methods. Afterwards, the process is executed while continuously monitoring key performance indicators (KPI). Typical indicators include the average time to complete a customer request, the lead time for orders, or the annual energy consumption. Such indicators are then analyzed, and possible improvements and bottlenecks in the process are identified, which in turn lead to a redesign of the process. The cycle is closed by reiterating the steps starting at the modeling phase.

In general, BPM spans over different functional areas of an enterprise. Different areas may use several systems for automating their individual role in a process. Thus, the typical IT landscape in an enterprise includes multiple information systems, such as production line automation, enterprise resource planning, supply-chain management, customer relationship management, and business intelligence and analytics. SOA

The different systems provide a large variety of services which can be combined to provide different applications and composite services. Such a *service-oriented architecture* (SOA) is the backbone of process automation and enter-

[1] A mote is a very small particle or speck.

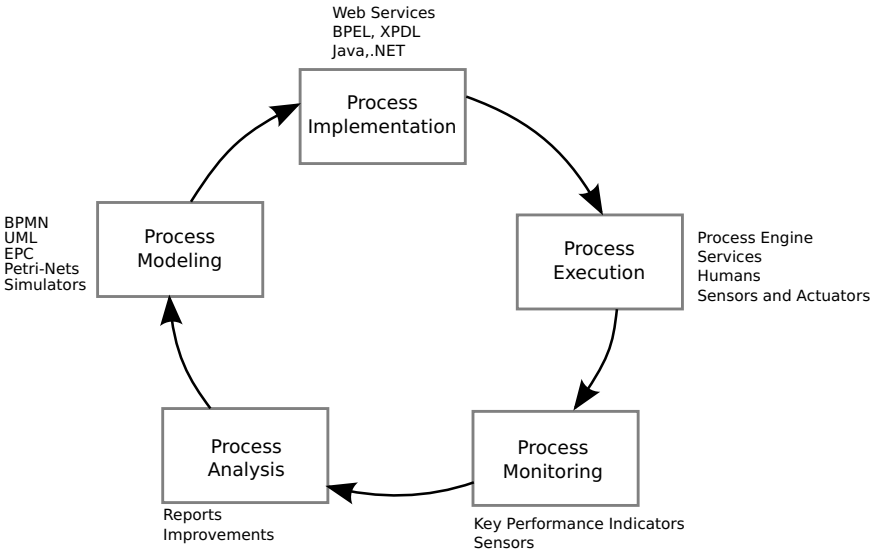


Figure 1.1: The iterative business process management cycle.

prise application integration (EAI). Managing all enterprise processes across such a complex IT landscape is a difficult task without standards for communication, interfaces, and data interchange. To enable interoperability among information systems, several open-industry standards have evolved. In this area, probably the best-known standard for communication are Web Services [183] with different Extensible Markup Language (XML) dialects for data interchange. An overview of the different standards, approaches, and architectures for business process management is given in [67, 179].

BPMS

Business process management suites (BPMS) provide enterprises with the required technical support for the entire life-cycle of processes. Figure 1.2 shows an architecture overview of a BPMS.

Firstly, BPMS assist business analysts in describing and documenting the blueprint or model of processes using graphical editors. Different wizards can assist them in creating Web-based data input forms. Secondly, the suite integrates all data descriptions, model blueprints, forms, and services in a shared repository where they are accessible to process users, creators, and owners using appropriate access control. In addition to model descriptions, stub templates for services can be created. The templates serve as basis for the manual implementation of services.

Given the specification of all processes, and provided all services have been implemented, a so-called process engine is used to centrally automate and control the execution of processes. Automatically generated Web portals offer different views over the process and allow users to interact with the process via forms. Using a similar interface, process owners can monitor progress and KPIs. In addition, different simulation and analysis tools help process owners identify possible problems and ways to improve the efficiency of processes. Given the

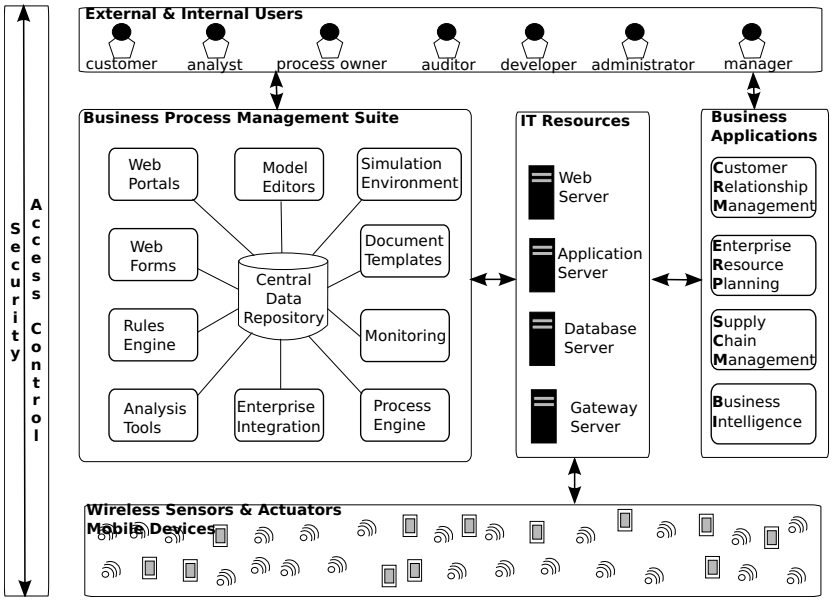


Figure 1.2: Schematic illustration of the architecture of a business process management suite. The components addressed in this thesis are the i) WSN; ii) the gateway, Web and application servers; iii) the model editor; iv) the simulation environment; and v) the Web-based portals. The remaining components are shown to offer a complete overview, but will not be discussed further.

complexity of a BPMS, a complete solution requires a substantial amount of resources, typically running on large server infrastructures.

The variety of BPMS offerings is large and comprehensive solutions are available both from major middleware vendors such as the IBM BPM [71], SAP NetWeaver BPM [147], and Oracle BPM [130] as well as from open-source systems like the JBOSS jBPM [80]. These solutions represent the state of the art for a comprehensive integration of human-centric processes on a large enterprise scale using the SOA paradigm. A recent market comparison and overview of current solutions can be found in [144].

1.3 Business Requirements

Business processes are designed and modeled by business analysts and consultants, who are domain experts in their field but not necessarily have a background in IT. Automation of business processes is where the complementary abstraction levels of business and technical models meet. The main business requirements with respect to process automation can be defined as follows:

- **Alignment** offers visibility over business-relevant behavior across the entire process implementation and execution.
- **Flexibility** means agility in composing and reusing existing processes models or execution platforms to meet new business requirements and regulations.

- **Confidence** means that the alignment of the process implementation and execution with the corresponding model has been validated and verified.
- **Efficiency** ensures that automated processes are improved and the costs of execution are reduced by the automation.

The above definitions will be used throughout this thesis when comparing different approaches for business process implementation and execution. The same definitions serve as a basis for establishing the requirements for the WSN run-time platform and development environment.

1.4 Classical Process Implementation

Using a pyramid hierarchy, Figure 1.3 depicts the classical approach to automating a business process based on a software implementation. Throughout this thesis, we will refer to the pyramid levels as *business process abstraction levels* [45]. The different approaches for automating business processes apply different techniques at each level. The approach and techniques being treated in a specific section will be highlighted in grey in the pyramid.

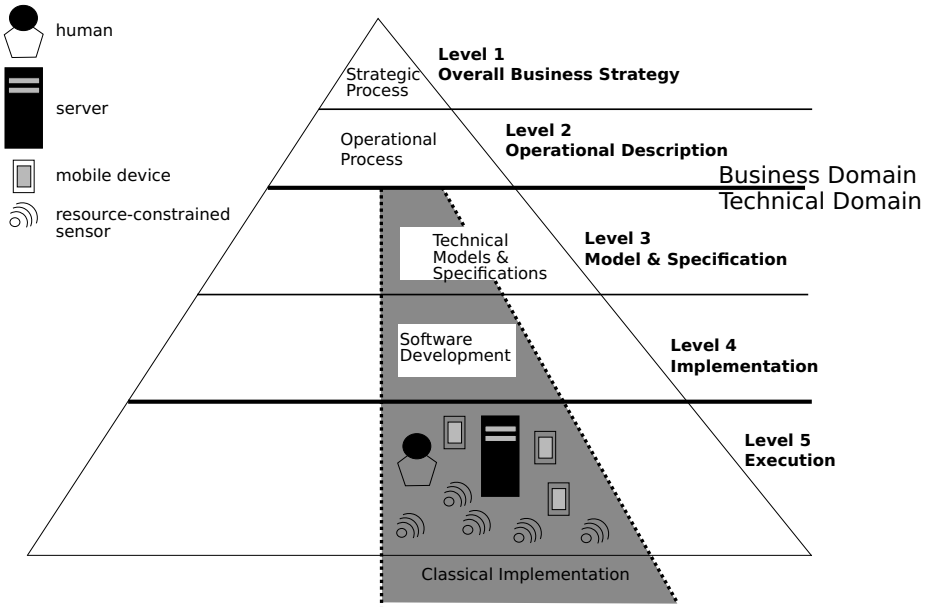


Figure 1.3: Business process abstraction levels [45] and the classical approach to process automation.

The first three levels of the pyramid are similar to the classification introduced by Silver [156]. Level 1 corresponds to a descriptive model, Level 2 represents an analytical model, whereas Level 3 provides all details required for execution.

At Level 1 business analysts start by laying out the strategic business process and its goals. At this level, the focus is on the core activities. The resulting process description represents the starting point for automation using a software

implementation. The strategic process is further refined at Level 2 into an implementation-independent operational process. This refinement describes the details required for humans or IT systems to perform their daily work.

Subsequently, business requirements from the operational abstraction level are communicated to IT professionals. These include the business-relevant behavior of WSN. To properly elicit all requirements, Level 3 focuses on building technical models and specifications to describe the systems required for the execution of the business process. At the next stage, in Level 4, software developers use the technical models to implement the actual process either manually or by using code-generation techniques. Finally, Level 5 represents the actual deployment and execution of the process.

1.5 Problem Set

This classical approach to process implementation poses a set of problems with respect to the main requirements for business process automation.

Alignment

Business process models are not always aligned with the corresponding implementation. The implementation usually deviates from the intentions and expectations of the business analysts because cross-domain communication is prone to misunderstandings. This deviation renders the respective business process the less effective the larger it becomes. Because WSN impose severe resource constraints, business-relevant behavior is either deeply embedded into platform-specific implementations or information from the WSN is propagated to back-end systems for further processing.

On the one hand, platform-specific implementations for business-relevant behavior are a dangerous foundation because they obstruct visibility over the process and they hinder the reuse of existing process models. That is, the process could be incorrect or not-compliant with regulations. On the other hand, delegating all control to back-end systems leads to process inefficiencies because the response time is increased, costly network traffic is generated, and only loose control loops over the processes execution are possible.

Conversely, if hardware wears out or is upgraded, the software implementation must be modified and migrated accordingly. These modifications are typically not reflected in the original business process model and may even be counter productive. Because the strategic and operational abstractions of the business process are no longer synchronized with the underlying implementation, the key performance indicators may provide biased data and cannot be used as a solid basis for process analysis.

To avoid these pitfalls, implementations should follow the business process and not vice versa [190] and should be generated automatically from the corresponding business process models. This approach ensures that the process execution is aligned with business requirements and provides transparency of the business-relevant behavior across the entire process.

Flexibility

Adapting to changing business requirements and new regulations is a costly and slow process. New requirements trigger modifications to the model of a business process, which in turn necessitates the adaption or even recreation of parts of the implementation, which is expensive and time-consuming if done manually [16]. As a consequence, business processes may be too slow in adapting to changes of requirements or business facts.

A flexible process execution environment is required that enables business to be agile with respect to changes in a highly dynamic global economy. In this respect, reusable components and process models which are hardware independent are crucial for accelerating process implementations or migrations to new or existing WSN infrastructures. The process execution environment should provide integration with enterprise systems, facilitate the composition of new processes using existing building blocks, and disseminate changes with minimal downtime.

Confidence

Incongruities in process implementations are often detected only in the execution phase. To increase the confidence level in the behavior of processes, models should be validated and verified using simulations, and tested and debugged prior to real deployments.

Such simulations prevent the propagation of communication misunderstandings into process implementations and allow business analysts to visualize a functional system early on. In addition, constantly monitoring the behavior of deployed processes using either online or off-line techniques increases the confidence that the behavior of the process execution corresponds to the business goals.

Efficiency

Efficient execution is a non-functional requirement that is not captured in business process models because they focus on functionality. However, an efficient platform with reduced power consumption, minimal resource usage, and a low execution time is necessary to fulfill the main goals of process automation: improving process efficiency and reducing cost.

For business processes distributed over a wireless sensor network efficiency is often ignored or entirely traded off for business alignment and flexibility. However, such a trade-off should not be the norm. The WSN execution platform should optimally and automatically manage power, use minimal resources, and efficiently execute the process on resource-constrained sensor nodes.

1.6 Thesis Goals

This thesis focuses on modeling and executing processes on resource-constrained WSN which meet the business requirements set forth in Section 1.3. The goals of this thesis can be summarized as follows:

- To align business processes with their WSN implementation to allow transparency and better visibility of business-relevant behavior over the entire process.
- To support business agility by composing reusable business process models that are independent of the WSN hardware platform to allow flexibility with respect to changing requirements and regulations.
- To enable an efficient process execution in terms of energy consumption and resource usage for the WSN platform to meet the business efficiency requirement by ensuring that the costs of a WSN implementation do not offset the automation benefits.
- To simulate and debug an actual WSN implementation from the business abstraction level to validate the behavior of a process model prior to its execution. This increases confidence in the alignment of the business process model with the corresponding implementation.

1.7 Approach

To address the problems inherent to classical process implementation, we propose to exploit a widely-used business process modeling language as the interface between business analysts and software developers. We then introduce code generation to turn process models into executable code. Process models can be refined by additional models, while commonly programmed components form the base case of this recursion. These components and the code generated from the models constitute the final application. Software developers can decide which levels of abstraction should be modeled and which should be programmed. At the topmost levels, business analysts, who represent our target user group, model the business process in a familiar business process notation, whereas software developers relieve them from mastering distributed computing problems by implementing all models and components on the lower levels. In our approach, instead of being just a documentation blueprint, a process model actually defines the execution of the respective business process and can be simulated and debugged directly in the model editor. Moreover, our tools must be able to generate efficient code that is able to execute on resource-constrained WSN devices.

A major benefit of a modeling or programming abstraction is the concealment of lower-level details. Without being forced to understand and master all those details, a user of the abstraction can create and maintain applications faster and more intuitively. This benefit is compromised if the application contains an error and there is no debugging support on the application's level of abstraction. In such a case, the user has to infer the error in the application code from the details of the code generated. Many WSN programming abstractions suffer from the lack of debugging support [114], which we believe hinders their adoption in practice. Thus, we emphasize the importance of testing and debugging business process models on the same level of abstraction using a simulation environment before their actual deployment. This is a best practice in finding problems or incorrect specifications and provides confidence that a modeled process will behave according to the business intentions.

1.8 Challenges

The challenges in reaching the goals set forth by this thesis can be summarized as follows.

- The first challenge is finding a widely-used modeling language that primarily addresses the business domain but at the same time is understandable by an IT audience. The language should have sufficient expressive power, and must be precise enough to allow direct execution.
- The second challenge is to model WSN aspects without extending the selected business-specific modeling language. Thus, a wide business audience can be reached, and detrimental effects on the execution semantics are avoided.
- The third challenge is meeting the business efficiency requirements and tight resource constraints of WSN while enabling a flexible reuse of business process models as building blocks which are independent of the hardware platform.
- The fourth challenge is integrating the simulation and debugging of process models into tools familiar to our intended business audience.

1.9 Methodology

We start in Chapter 2 by exploring different languages and techniques for business process modeling and execution, respectively. Because business process modeling is a domain-specific instance of the more general approach of model-driven system development, we analyze those approaches that address WSN from a technical perspective. The aim of this analysis is to find an existing business modeling language which addresses our first challenge. Furthermore, we assess the extent to which existing approaches solve the remaining challenges set forth by this thesis. We then argue why our approach advances the state of the art and describe our design decisions. Then, in Chapter 3, we show how the main WSN-specific aspects can be captured and fully specified without extending or adapting the execution semantics of the selected modeling language. Next, we focus on the transformation of business process models into event-based source code which can execute on major wireless sensor platforms. We describe our pattern-based compilation approach in Chapter 4. Subsequently, we consider the set of integrated tools required by our model-driven approach in Chapter 5. We then analyze the individual resource usage of platform operations in Chapter 6, which leads to further considerations and optimizations to reduce energy and memory usage. These optimizations address both the underlying run-time platform and the code generation. In Chapter 7, we evaluate the proposed modeling approach against the previously introduced business requirements. For the evaluation, we use our implementation of the required tools, which include the presented optimizations. Finally, we conclude and present future directions for our work in Chapter 8.

Chapter 2

Model-Driven Development

Vision is one of the most evolved human senses and humans tend to think in pictures which can often convey a large amount of information in a compact form. Consequently, the use of graphical models is a widely employed technique that facilitates reasoning about system behavior and fosters communication among different participants.

The software development approach which uses graphical models not only for documentation or as a means to facilitate reasoning but primarily to generate a final executable application is referred to as model-driven development (MDD). Model-driven approaches are applied to various domains to increase the level of abstraction, improve productivity of developers, simplify application development, offer a better system overview, formally prove correctness of systems, or address an audience not necessarily skilled in IT. Our aim is to empower business analysts to fully control processes executing on embedded devices using a familiar graphical notation.

According to their intended audience, we classify modeling techniques into two main categories: business and technical. The business audience includes business analysts, consultants, process owners and managers, whereas the technical audience encompasses scientists, engineers and software developers.

Models in the business domain focus on mapping the main steps or structure of a business process with its entities and their respective interactions. The goal is to identify ways to improve the efficiency of a business process. Improvements are usually achieved by automating certain process steps or even completely redesigning the entire process.

Models in the technical domain focus on architecture and system decomposition, timing and synchronization, data structures, signal filtering, and algorithms. Here, the goal is to formally specify the structure and behavior of hardware and software systems to facilitate or even automatically generate a complete implementation.

Automation of business processes is where the complementary views of business and technical modeling meet. Most modern business processes require automation, which in turn is based on technical models used for the implementation. As a consequence, the description of a business process must enable a technical audience to extract the correct requirements for the implementation.

Our focus is the automation of business processes which target embedded devices and in particular resource-constrained wireless sensor networks. Thus,

we first analyze the business process modeling and execution techniques used for automating business processes in general. Next, we review various technical modeling languages and the respective model-driven solutions that address WSN. We then analyze approaches which integrate WSN with business processes and show that there is still a gap between the specification of process models and their execution on resource-constrained WSN.

2.1 Business Process Modeling

Typical business process models describe the steps required to reach a strategic business goal or the sequence of tasks or operations which guide humans in their daily work. A flow-chart is a type of graphical representation which can capture such basic aspects and is widely familiar to the business audience. Our focus is not to design the perfect graphical modeling language. In the following, we analyze existing business modeling notations which already are tailored for the business domain, are widely-used by the respective community, and have proved their applicability in practice.

Our main search criteria is a graphical modeling language which is understood primarily by business analysts and at the same time can be precise enough to convey execution requirements to software developers for implementation. In a first iteration, certain business process details such as exception paths, different events or the flow of data may even be completely left out from the model diagram. Ideally, models can be further refined to a level where they can be either directly executed or transformed to executable code. In essence, such fully refined models contain technical elements which might not be completely understood by business analysts. Nevertheless, if the modeling language provides the same basic notation, business analysts can grasp the core structure and reason about the behavior of the process. A common notation helps communicating requirements in a precise way. Refining such models even further may require some technical background. The missing puzzle piece which enables models to execute is the information about service invocation, data inputs and outputs, communication protocols, and message encapsulation.

2.1.1 Business Process Model and Notation (BPMN)

The Business Process Model and Notation BPMN [129] is a graph-based language which addresses the needs of the business domain. In collaboration with several industry partners, the notation was standardized under the Object Management Group (OMG). Throughout this work we will refer to the latest version of the standard, namely, BPMN 2.0, which adds significant improvements to the previous versions, including new event types, execution semantics, and a well defined XML serialization format. Today, the BPMN language is widely used by business analysts, consultants, and executives as a lingua franca and rapidly gains popularity with the IT community [45, 156].

BPMN integrates in a coherent and compact view all business process aspects regarding control and data flow, entities and resources, as well as events, communication, and the business organizational structure. Moreover, the execution semantic of BPMN diagrams is well defined by the standard which makes the

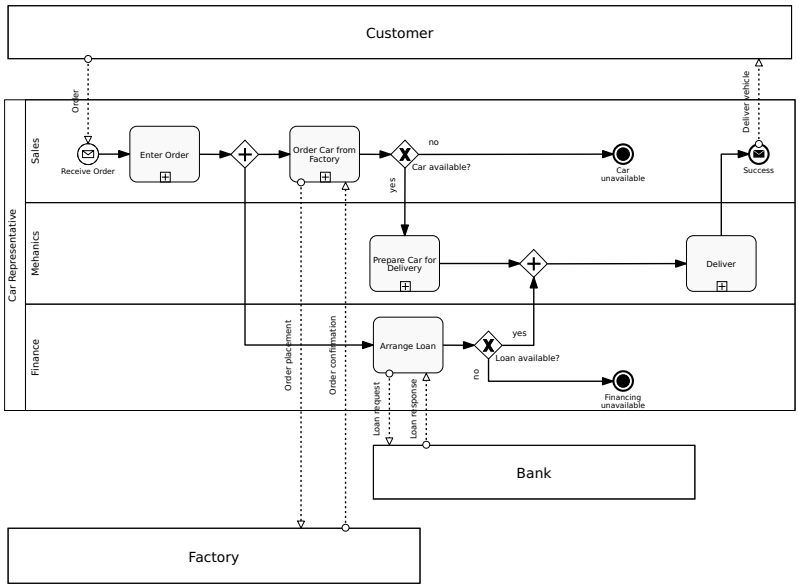


Figure 2.1: A descriptive BPMN process model for ordering a car from a dealership (adapted from [156]), which documents the sequence of manual tasks and events, irrespective of the technology behind them. Concrete examples of processes using WSN technology will be discussed in the next chapters.

graphical notation suitable for both direct execution as well as transformations to executable languages.

Figure 2.1 shows an example business process for ordering a car, from the perspective of the car dealership. The process involves the customer, the sales representatives as well as the car factory and bank. The focus is on the car sales representatives whose process is shown in detail over three functional units: sales, mechanics, and financing. The car order is successful and the customer is informed only if both a loan is available from the bank and a car is available from the factory. Otherwise the process terminates. This example describes a classical process which is mainly performed manually. Some parts of the process may be automated. For example, once the sales department enters a car order an automatic message is sent to the factory. The description focuses on the individual activities and events rather than the technology behind them. Examples of processes where WSN technology is involved will be discussed in detail in the next chapters.

A theoretical analysis of BPMN in terms of expressibility and suitability for modeling business processes is discussed in [181]. In terms of expressibility of control flow, BPMN is a powerful language. Out of the twenty well established workflow patterns described in [170], BPMN lacks support only for two, namely, “milestone” and “multi-instance without a priori run-time knowledge” [181]. Formally BPMN is a hybrid language which incorporates concepts from both Petri-nets [139] as well as communicating extended finite state machines (CEFSM) [15, 20].

2.1.2 Event-driven Process Chain (EPC)

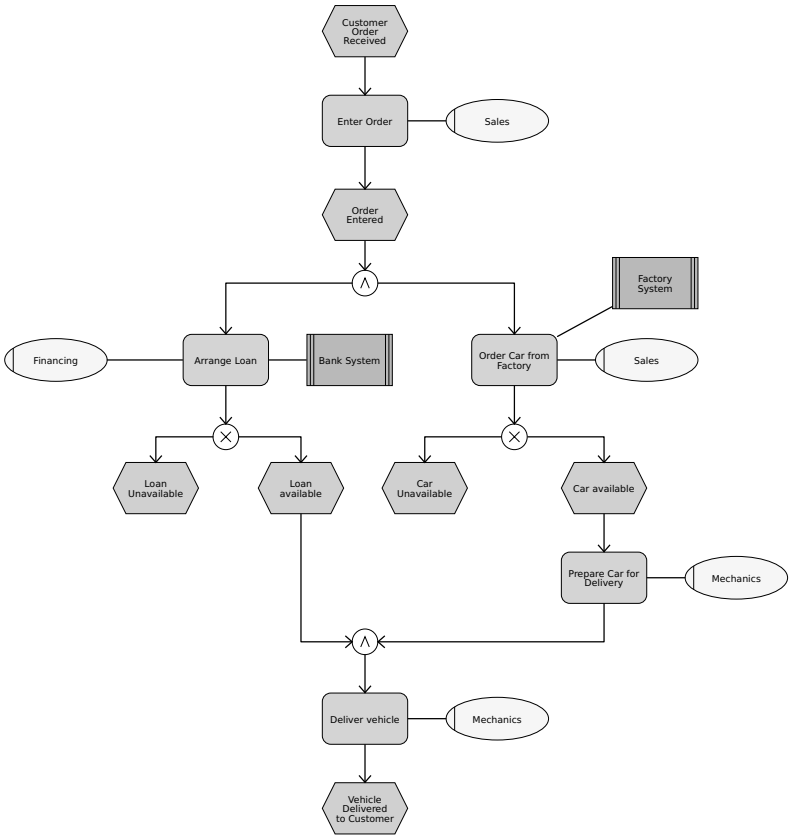


Figure 2.2: A business process model for ordering a car, described using the EPC notation. This process is an equivalent to the model in Figure 2.1.

One of the first graphical notations used to describe business processes are so called event-driven process chains (EPC). The notation was originally designed to support modeling for the SAP's R/3 system in the ARIS process management methodology [31, 83, 148, 149]. The notation can describe control and data flow, people and resources involved in the process and as the name suggests has native support for events. If one only considers the core aspects of control flow and events, the underlying formalism for EPC are Petri-nets [139] as shown by the transformation provided by [109].

The initial focus of the EPC notation was not on executable processes. However, the tools and methodologies developed around EPC allow such diagrams to control and execute processes. In terms of control flow, the EPC notation lacks support for almost half [108] of the established workflow patterns [170]. EPC diagrams tend to contain more symbols than their equivalent BPMN models [87].

Figure 2.2 shows an EPC equivalent of the business process for ordering a car from Figure 2.1. In contrast to the previous BPMN example, the organizational

structure for the car dealership is not explicit but can be modeled as part of a separate diagram. Also, the interaction with the external entities for the bank and factory is modeled as a system association which assumes the orders to the factory and the loan inquiries occur through the respective IT systems. The EPC diagram introduces an `Order Entered` event which is not present in the BPMN diagram. Furthermore, events have no visual type and are only distinguishable by their name. Hence, it is not evident that certain events will terminate the process, such as the `Loan Unavailable` event.

The EPC notation is still in use today to capture business processes, however, the tendency is to migrate to the new BPMN standard and some work exist to partially automate this transition [33].

2.2 Process Execution

A process-centric model-driven development approach [45] focuses on further refining the details of a business process from the Level 2 operational description down to a complete Level 3 executable specification. This approach is highlighted in Figure 2.3. This refinement is typically performed using the tools provided by a BPMS. Process-centric implementations of business process are gaining momentum and many vendors and business consultants encourage this trend.

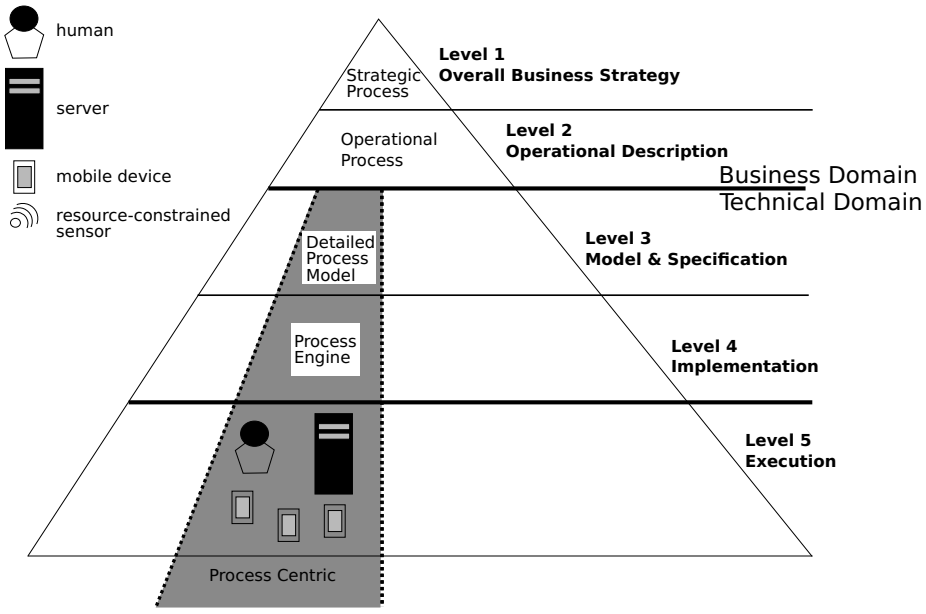


Figure 2.3: The process-centric approach for automating business processes.

A core component of a BPMS is a *process engine* which is responsible for the actual process automation. The engine executes a machine-readable model specification and orchestrates the interaction of systems. process engine

The Business Process Execution Language (BPEL) [124] represents one way of describing a process specification in machine readable form. BPEL is an

open-industry XML standard intended for execution by a process engine. Even though BPEL is not a graphical notation, several vendors offer a graphical representation in their modeling tools to assist the development cycle. Several other proposals which define formats for process execution languages exist such as the XML Process Definition Language (XPDL) [182] or Yet Another Workflow Language (YAWL) [164]. With the emergence of the new BPMN standard which provides both a graphical notation for humans as well as a machine readable execution format, the BPEL standard is losing some ground.

One of the main advantages of using a process engine are audit trails as well as monitoring capabilities and separation of concerns. The reusable execution logic is decoupled from the business logic. The process engine takes care of synchronization and concurrency, the invocation of Web Services, handles the manipulation of data and messages, is able to recover from errors, and supports concurrent processes with long or short living execution scenarios.

Typically a process engine is tightly integrated into the middleware provided by a BPMS. The integration is at times proprietary which makes it difficult to extract only the core functionality of the process engine, which is controlling the execution of a process.

embedded process engines

Because BPMS are monolithic and heavyweight they do not meet the tight resource requirements of embedded devices. Moreover, the functionality of a process engine cannot be separated from the corresponding BPMS. As a consequence, some light-weight process engines have been proposed. An example is ePVM [176] which address process-centric applications. Interestingly enough the process description is not specified using the BPEL standard but JavaScript. This allows the ePVM engine to be embedded into mobile devices.

Other examples of light-weight process engines implemented in Java which specifically address mobile phones and PDAs are CiAN [152] and Sliver [61]. However, the minimal requirements for a process engine are at best addressing the class of devices powered by 32-bit MCU with significant amounts of persistent and volatile memory.

2.3 System Development

The development of complex hardware and software systems required by today's dynamic economy would be a cumbersome task without the aid of graphical models. Because such models are used to generate an implementation they must have a precise semantic and formal definition. There is no room for interpretation when building a new hardware chip or a software controller for an air-craft turbine. The step from graphical models to executable applications is a similar paradigm shift as the compiler which transforms high-level languages to machine code.

In the following, we focus on the model-driven software development for WSN applications. As such, we discuss the major technical modeling languages available for developing embedded applications and the solutions which directly address WSN. Figure 2.4 highlights this approach for implementing the technical systems required by business process.

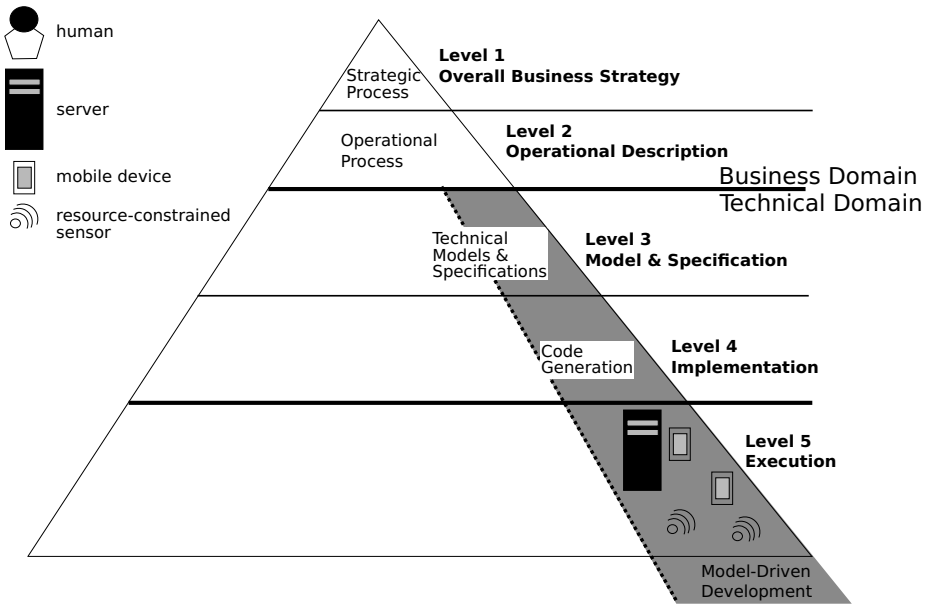


Figure 2.4: The model-driven development of technical systems which enable the automation of business processes.

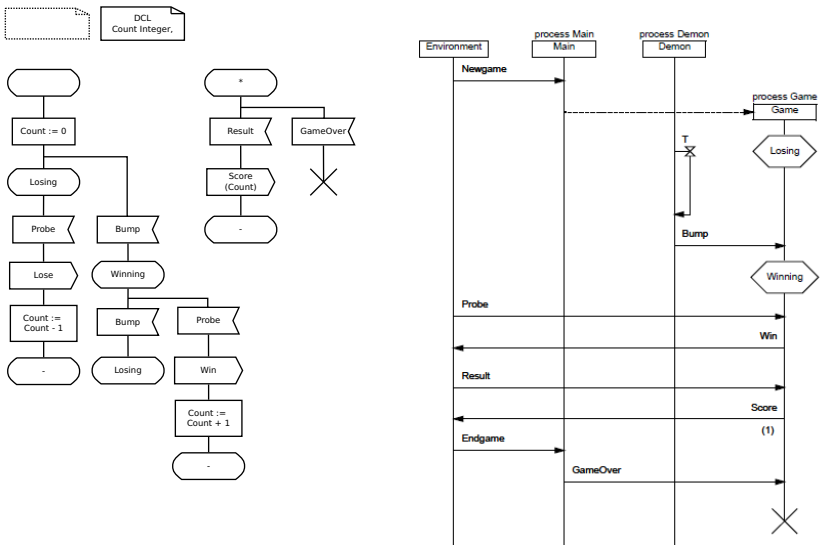
2.3.1 Specification and Description Language (SDL)

A well-known open-industry standard modeling language is the Specification and Description Language (SDL) [34] which is widely used in development of event-driven communication systems. The underlying formalism of SDL are communicating extended finite state machines [15, 20].

The SDL language has both a graphical as well as a phrase representation. Because its execution semantics are formally complete, SDL can be used for code generation. In SDL, the environment contains the set of all systems which are further modularized using blocks, processes, and procedures. Communication is modeled as signals which route inputs and outputs to and from processes. A workflow-like description is used to describe the activities inside a process as well as reusable patterns encapsulated in procedures. The definition of an *SDL process* is the one commonly used in operating system terminology, i.e, a program which executes and which is started and scheduled by the operating system. Thus, SDL processes should not be confused with business processes as defined in Chapter 1.

Sometimes used in conjunction with SDL models and often part of technical hardware specifications and description for communication protocols are Messages Sequence Charts (MSC) which show the ordering of exchanged messages between processes and systems for different use cases.

Figure 2.5(a) shows an SDL process description for a `Game` [75] behavior. The process controls the rules for how users can play against a `Daemon`, described by its own process. The game starts by initializing the score counter value to `0` and the process enters the `loosing` state. Subsequently, the process waits for two possible input signals or events: a `probe` signal from the user, and two `bump` signals from the `Daemon` process. The user wins the game and the score



(a) Behavior of the `Game` process (b) Message sequence chart for a game run

Figure 2.5: Example SDL diagrams (adapted from [75]).

counter is increased if the state of the game is probed between the `bump` signals from the `Daemon`. The state of the game can be queried at any time using the `result` signal.

Figure 2.5(b) shows an MSC which depicts the exchanged signals for a game run where the player wins against the `Demon`. This is because the `probe` signal arrives before the second `bump` signal. Finally, the user ends the game after querying the score.

industry solutions

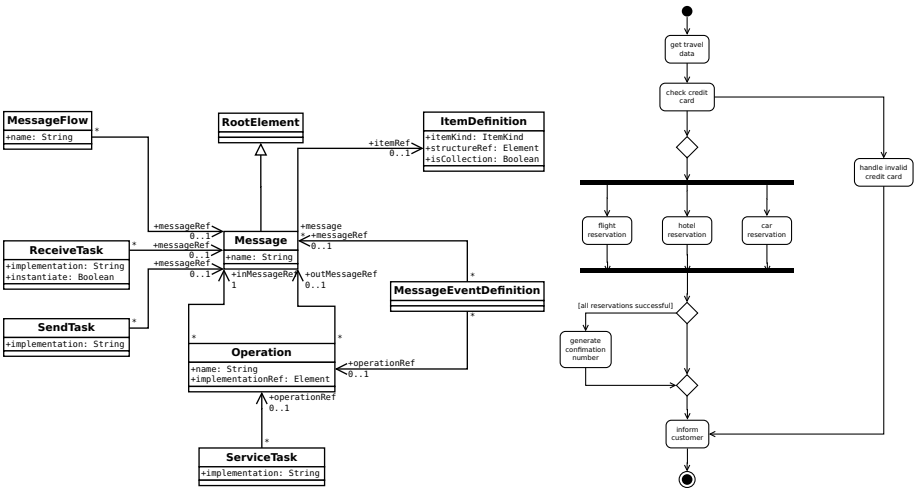
The IBM Rational SDL and TTCN Suite [76] is an industry solution which implements an SDL graphical editor and analyzer together with a simulation environment for debugging models. The solution can generate executable C/C++ code for embedded systems. The simulation environment acts as an operating system where SDL processes are created, communication is delayed, and the system and environment state can be queried and modified.

WSN

SDL is typically used for the design, verification and implementation of communication protocols by telecommunication companies. The formalism and simulation tools provided around the SDL methodology have been exploited by the WISENES [89] design and simulation framework for WSN. An efficient automatic code generation with a tight integration between the SDL run-time and the execution platform for WSN is explored in [173] with a target platform based on a 32-bit LEON2 processor.

2.3.2 Universal Modeling Language (UML)

Probably the best-known modeling language is the Universal Modeling Language (UML) [126] which is a generic set of open-industry (OMG) standard graphical notations that capture different system aspects. A system's architec-



(a) Class diagram (adapted from [129]) (b) Activity diagram (based on [74])

Figure 2.6: Example UML diagrams.

ture is captured in package, component, deployment, and composite structure diagrams.

In practice, the UML class and object diagrams are frequently used to describe data structures and their relations. Figure 2.6(a) shows an example UML class diagram for the data model and relationships of the BPMN `Message` class. The class inherits all the attributes from the basic `RootElement` and is referenced by the classes for sender and receiver as well as the corresponding message flow.

In UML, the behavior of a system is captured using different perspectives. Activity diagrams describe the control flow, state machine diagrams show the system transitions, while interaction diagrams specify the communication exchanges and timings. The UML interaction diagrams are very similar to the MSC diagrams from SDL.

UML activity diagrams can be considered as the precursor for the BPMN language. However, for an equivalent BPMN description of a single business process, UML requires multiple models which cover different aspects and different involved entities of the business process. Due to historical reasons, the UML activity diagrams are still in use for documenting business processes. Figure 2.6(b) shows an activity diagram which describes the business process for a travel booking [74] which requires a hotel, a car and a flight reservation. If any reservation fails, or if the provided credit card information is not valid the process terminates unsuccessfully.

industry solutions

An established industry solution for UML-driven development and automatic code generation for embedded systems is the IBM Rational Rhapsody [73]. The solution includes a simulation environment where annotations in the generated code are used to simulate the system behavior and allow developers to interact with the modeled prototype. Examples of open-source solutions for UML development providing model transformations and code-generation are Papyrus [135] and MOFScript [113] which are based on the Eclipse Modeling Framework

(EMF) [162].

WSN More recent work in the WSN space [47, 48] uses the notation of UML activity diagrams and the Papyrus generation framework to control wireless SunSPOT [131] nodes with a 32-bit ARM core. The GAF4WSN framework [2] allows developing WSN applications for target platforms based on an embedded Linux distribution. The framework uses the UML notation for class diagrams to define data types and the UML use cases notation in conjunction with SDL diagrams for specifying behavior.

2.3.3 Data Flow Models

Data-flow models focus on data and how it is processed by systems as opposed to control-based approaches typically used in imperative programming languages. In data-flow programming, graph elements become active and execute when all their inputs are available and after processing they produce respective outputs which in turn trigger further processing steps.

Systems are thus decomposed into functional block which work on input signals and produce output. Blocks are interconnected to compose functions such as filters, integrators, multiplexers, de-multiplexers. These produce results which may be visualized or further used to control or communicate with systems. Flowing of time is also specified using blocks which define the rate at which output is produced.

LabVIEW

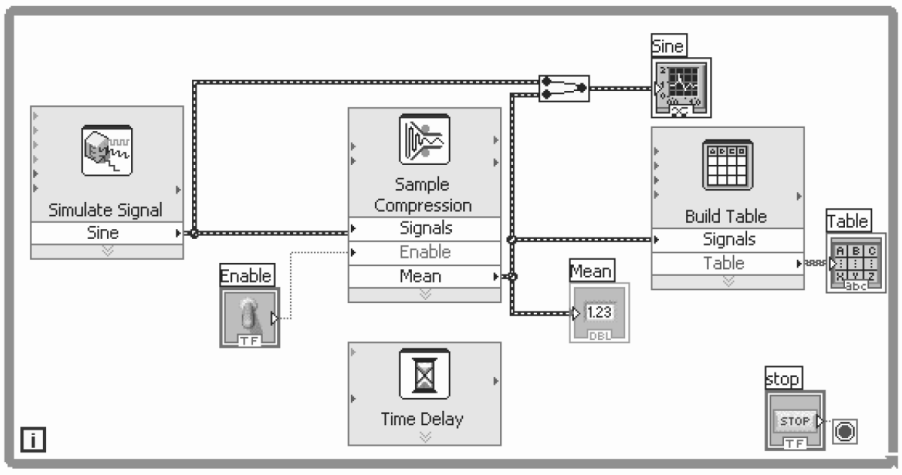


Figure 2.7: LabVIEW model example (adapted from [120]).

The well-known LabVIEW [119] solution from National Instruments (NI) provides engineers and scientists with a graphical data-flow programming language called G. The solution integrates a development environment with simulation and visualizations capabilities which imitate physical instruments such as oscilloscopes and multimeters. The most common use case scenarios for LabVIEW are: acquiring and processing measurements, controlling experiments and

instruments as well as embedded hardware design such as field programmable gate arrays (FPGA) using the hardware description language (HDL).

From the graphical specification, LabVIEW can generate both C code or dynamic link libraries which can be integrated with other systems or development tool-chains. Integration between the LabVIEW environment with different measurement devices is achieved through specific drivers.

Figure 2.7 shows an example LabVIEW model [120] which describes an analysis of a simulated sinusoidal signal at a sample rate controlled by `Time Delay`. The signal can be compressed and both the original and the processed signal can be displayed in the same graph. The compression is only active if the respective Boolean knob `Enable` is set. In addition, the mean value of the signal can be wired to the output of the compression and displayed graphically. Last, the compressed data samples are saved in a table for later processing.

The graphical programming and code generation capabilities of LabVIEW have been exploited by the Cubix [51] platform which runs an embedded Linux on an 32-bit ARM7 processor. More recent developments [121] allow to graphically program and control NI-proprietary WSN devices [122] using a subset of the features provided by LabVIEW. WSN

Simulink

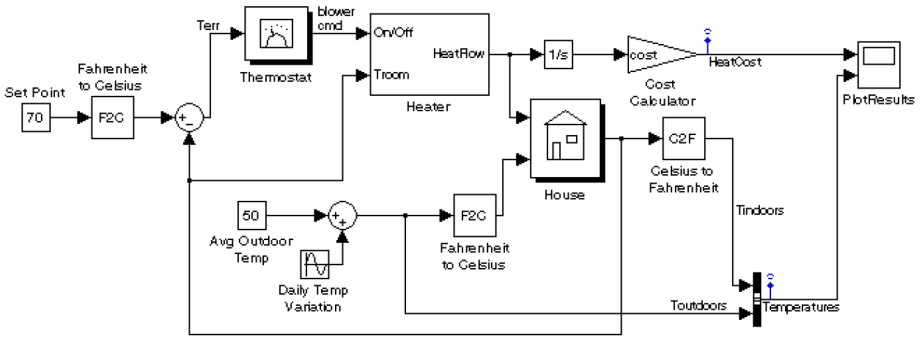


Figure 2.8: Simulink model example (adapted from [102]).

The popular MATLAB [100] technical computing solution from MathWorks provides several extension packages for describing and simulating system behavior as well as generating executable code. One such extension is the proprietary data-flow modeling standard Simulink [102]. Another extension to MATLAB is Stateflow [103] which is a graphical representation of system behavior using states and transitions as well as blocks which can be defined using Simulink. The Real-Time Workshop Embedded Coder [101] can generate C/C++ code for embedded systems from corresponding Simulink and Stateflow models. Because such models are usually executed in the simulation environment or are used to generate code certain naming rules have to be respected.

The Simulink environment mainly address scientists and hardware designers which can model complex signal processing, as well as controllers and communication systems. Mathematical libraries together with simulation, statistical

reports, and graphical visualization are available and tightly integrated with the MATLAB framework.

Figure 2.8 shows an example for a thermal model [102] which allows to test the influence of temperature variations on the control loop of a house thermostat. The temperature variations of the outside environment are modeled using a sinusoidal signal. The diagram consists of several subsystems: thermostat, heater, house, and the temperature degree conversion functions. To monitor the efficiency of the control loop, the heating costs together with the indoor and outdoor temperature are plotted in a graph. The house thermostat is initially set to 70 degrees Fahrenheit which are then converted to degrees Celsius. Based on the difference in the desired temperature and the measured house temperature the thermostat decides whether to send a command to the heater.

WSN

In the WSN space, the features provided by Simulink have been exploited and integrated into an advanced framework for modeling and simulating [116] applications. The framework supports the complete development cycle and targets 8-bit and 16-bit platforms running the TinyOS [68] and MANTIS [13] operating systems.

2.3.4 Domain Specific Languages (DSL) for WSN

So called domain specific languages (DSL) are used to allow scientists to seemingly control the behavior of instruments and systems and conduct experiments without dealing with implementation specific details. Such languages are typically graphical and provide a set of pluggable components which domain specialists can reuse to create applications.

A popular open source operating system for wireless sensor networks is TinyOS [68]. The system is based on a component architecture where building blocks can be substituted and wired to create an application. To help the wiring of components and to support the event-based programming required by WSN, TinyOS uses a special C dialect called nesC.

GRATIS II [57] is a modeling environment which aims to ease the development of TinyOS applications by providing graphical tools to wire interfaces and components which are represented by blocks. The Generic Modeling Environment (GME) [54] is used to define a meta-model for the TinyOS components. This provides a good overview of the interconnections for complex TinyOS applications. However, such a component model is mainly used for static analysis of the wiring of interfaces and does not support execution in a simulation environment. Moreover, the corresponding models are platform specific which requires a deep knowledge of the target run-time platform. Another plug-in specific for nesC is provided by the CADENA [174] general purpose model-driven application development environment. This plug-in is similar to the tool provided by GRATIS II and can generate code templates and wirings for the TinyOS platform.

To support model-driven development of WSN, the approach in [95] introduces a DSL which abstracts from implementation issues. The DSL is first transformed to an intermediate model described by UML activity diagrams. These are finally translated using MOFScript to produce nesC code for an actual TinyOS implementation.

To address the requirements of WSN applications several other DSL and respective model-driven frameworks have been proposed. Viptos [26] and Flow

[123] are modeling environments for wireless sensor networks with custom graphical notations focusing on data flow. Their graphical representation is similar to Simulink and LabVIEW models. Their intended audience are domain scientists, system engineers, and software developers rather than business analysts.

The Abstract Task Graph (ATaG) [9] uses a macro-programming approach to describe WSN applications. These are described in terms of abstract data and how this is processed by abstract tasks which can be local or distributed as specified by model annotations. The tasks are generic and are only bound to a specific platform and respective sensor nodes in a compilation and deployment phase based on target system descriptors. The solution is only semi-automated as programmers are required to provide the implementation for the task functionality for a particular platform. Srijan [136] is modeling environment based on GME which allows to graphically specify a network configuration as ATaG macro-programs. These are then compiled to Java source code for the SunSPOT platform. Finally, developers specify the final implementation details for individual tasks. The methods around ATaG do not address platform specific optimizations and require a 32-bit ARM hardware configuration.

Other application domains benefit from DSL as well. For example, the Scratch [143] and LEGO MindStorms [90] graphical programming environments open the world of graphical programming of games and LEGO robots to an audience with little background in computer science.

2.3.5 Discussion

From a theoretical point of view all presented technical modeling languages are suitable for code generation. In the following discussion, we use the terminology from the Rhapsody solution [73]. However, the same principles are valid for the remaining solutions described in this chapter.

In general, model-driven approaches generate executable code which is independent of the underlying operating system (OS). Thus, the generated code uses a run-time environment or object execution framework (OXF) which abstracts the OS and provides reusable functionality such as event processing, message queues, memory management, inter-process communication, and thread synchronization. Using object-oriented design principles, the OXF can be reused for different platforms which reduces the size of the generated code. Based on this principle, the execution framework can be customized using inheritance to match the requirements of particular target platforms.

The interaction between the platform-independent OXF and the underlying OS is achieved through a thin set of primitives which provide an operating system abstraction layer (OSAL) which implements tasking services, synchronization services, communication ports, and timer services. In essence the execution logic is reused by means of a middleware. Porting the generated code to a real platform means implementing the functionality required by this platform-specific layer.

For platforms such as sensor motes customizing the OXF and implementing the OSAL is possible in theory but challenging in practice given the resource constraints of the lower-end spectrum of the platforms we target. In addition, some adaptation of the code generation process may be required for the actual platform.

From the presented model-driven development approaches in the WSN space only two frameworks, namely, [95] and [116] were able to reach the resource-constraints we are targeting. However, none of the existing solutions discusses neither energy consumption nor memory usage and they strictly address a technical audience.

In general, technical, graphical modeling languages are extremely valuable in the development cycle of software systems [62]. Furthermore, model-driven approaches have proven their applicability in practice [125] in the development of business-agnostic behavior such as algorithms, communication protocols, hardware design, and drivers. However, to communicate requirements to and from the business domain, technical languages lack support for business-centric concepts, and the clarity and graphical conciseness provided by modeling languages specific to the business domain.

2.4 Aligning WSN and Business Processes

Integrating and aligning WSN with business processes in enterprise applications is not an entirely new idea. Existing approaches [81, 160] focus on the SOA paradigm and advocate exposing smart devices and WSN as a Web Service either natively or through special purpose gateways which perform a protocol translation. Essentially, the capabilities of a WSN are thus augmented or Web-enabled and services can be discovered and composed [93] dynamically in an object-oriented manner. To facilitate such approaches, the Web Services 4 Devices Profile (WS4DP) aims to provide and efficient implementation for all aspects of the Web Services standard including communication based on XML and SOAP and support for service discovery. An alternative approach for incorporating devices into a business infrastructure is to use standard HTTP [58] technologies, such as the Representational State Transfer (REST) architecture [43]. A standard REST interface enables common Web browsers to interact with devices which are addressable as URI resources. Both approaches to exposing the services of a WSN are fully complementary to our work which does not impose a particular communication method.

In practice, both Web Services and REST still impose a significant overhead for the resource-constrained devices we are targeting. Consequently, [53] addresses the resource constraints of WSN by compressing the verbose SOAP/XML format used by Web Service and introducing a lean transport protocol (LTP). An even leaner approach is presented in [97], which allows expressing basic business rules using a custom phrase language. The rules are then efficiently encoded and disseminated in the WSN and interpreted at run-time by the sensor nodes which communicate deviations to back-end services. However, rules are reduced to a limited set of functions which must be implemented and interpretable by the underlying platform. In other words, the building blocks for rules and the rule interpreter are fixed.

A recently started European project, makeSense [36] aims to close the gap between business process models and WSN using an intermediate macro-programming abstraction. This abstraction would be reusable across different WSN applications and business processes. The macro-programming abstraction is based on a middleware layer akin to the TinyLIME [28] or logical neighbors [115] approaches. Our approach differs in two main aspects: modeling and

compilation. First our BPMN modeling style allows to minutely control even the behavior of individual nodes in the network. Second, we directly compile the process description to an executable application. Moreover, we are able to adapt the compilation to use specific communication APIs based on the needs of the application. In other words, our approach is not limited to a particular communication middleware.

In [53], BPEL is used in to graphically specify the execution of a process controlled on the server side which invokes the exposed Web Service provided by the WSN devices. The same approach is taken by the GWELS [52] environment which uses BPEL running on back-end servers to orchestrate the execution of workflows distributed in a WSN.

Even though the existing approaches allows WSN devices to expose their functionality as a standard service, the implementation behind the service is not fully aligned with the business process models. Changes in business requirements imply changes to the implementation of the corresponding service provided by the device. Often, business process models do not execute directly in the network but on back-end servers.

All presented approaches are not concerned with energy consumption nor execution efficiency. In addition they lack the support of an open-industry standard graphical notation which hinders the interchange and reuse of business process models.

2.4.1 A Gap Remains

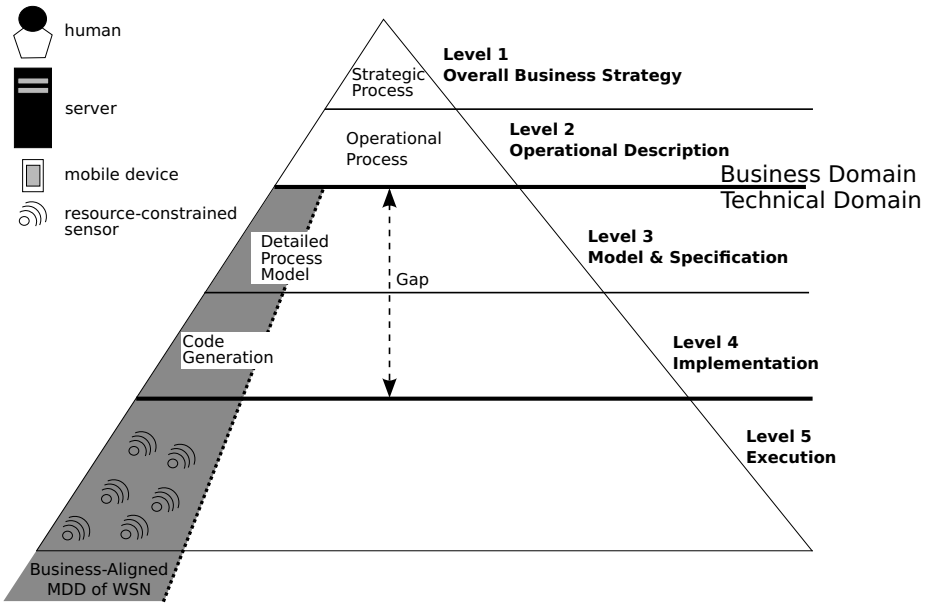


Figure 2.9: Business-aligned implementation of WSN using a model-driven approach to bridge the gap between the business and IT domain.

None of the existing solutions bridges the gap between the model descriptions provided by business analysts and the corresponding implementation of a

business processes for the resource-constrained sensor networks we are targeting. Figure 2.9 shows this gap between the business abstraction levels and the process implementation and execution. Bridging this gap in an efficient manner represents the main focus of this thesis.

On the one hand, although some of the technical modeling solutions may be adapted to accommodate the resource requirements, they focus on the classical business process execution and expect the business audience to learn and understand a technical language and the associated tools. On the other hand, solutions targeted particularly at the business domain use the process-centric approach but require too many device resources and lack the ability to specify and control the behavior of resource-constrained wireless sensor networks.

Moreover, both the technical and business solutions do not concentrate on efficiency in terms of energy consumption nor resource usages. These factors are important in reducing both investment and maintenance costs of automated processes.

To bridge this gap, we combine the benefits of existing work and use an open-industry standard for modeling business processes in a familiar environment. We allow business analysts to specify reusable executable requirements which align the behavior of resource-constrained wireless sensors to flexible business needs. Our approach provides a unified view over all involved systems, supports simulation and debugging of business processes directly in the model view, and generates efficient code.

2.4.2 Our Design Decisions

As a modeling language for our business audience, we choose the new BPMN 2.0 open-industry standard to leverage the existing set of tools, user base, and best practices [156]. The standard specifies execution semantics which makes it suitable for direct execution by a process engines. BPMN is a suitable interface language between a business audience and software developers [156]. The language can provide different abstractions levels [45] which allow it to be understood by both sides, yet it can be precise enough to serve as a basis for execution.

Subsequently, we use a compiler to generate efficient code from a BPMN model which executes directly on the targeted sensor node platform. In essence, we generate a custom process engine for a particular business process model. For the resource-constrained sensor nodes which we want to support, a model interpreter like a (BPEL) process engine executing on the node itself is not a viable option. Moreover, there have been reports of difficulties in directly mapping BPMN to BPEL [175].

Furthermore, with our solution, different communication channels can use different protocols such as IEEE 802.15.4 or TCP/IP depending on the respective requirements to facilitate application integration. We are thus not dependent on a particular communication standard like Web Services.

Using this approach we are able to scale down to the required platform but we loose the ability to trace history and monitor the process execution in real-world deployments. However, this ability is fully retained in an accurate simulation environment. With appropriate techniques for capturing wireless traffic and run-time analysis and verification [146] some of the information may be inferred even for real-world deployments.

In contrast to [110, 159], we show that BPMN is expressive enough to capture all aspects for WSN applications without the need for graphical extensions. Extending the language with special symbols and annotations hinders the universal understanding and limits the communication capabilities between business analysts and software engineers, which is the main advantage of using BPMN. Moreover, the impact of language extensions on the execution semantics is not clear.

Summary

Having decided on suitable language is but a small step. At this point, we merely have a set of words, syntax, and grammar which can be used to write descriptions. Now, these elements must be combined in a meaningful manner and using a certain style to specify the details of executable WSN applications.

Chapter 3

Modeling WSN Applications using BPMN

Wireless sensor networks are a particular instance of distributed applications which use a shared wireless medium for communication, are reactive by nature, and potentially consist of a large number of actors. This chapter describes how the functional behavior of such applications can be mapped to existing BPMN concepts without extending or modifying the execution semantics of the language. In other words, we define a BPMN modeling style for WSN applications.

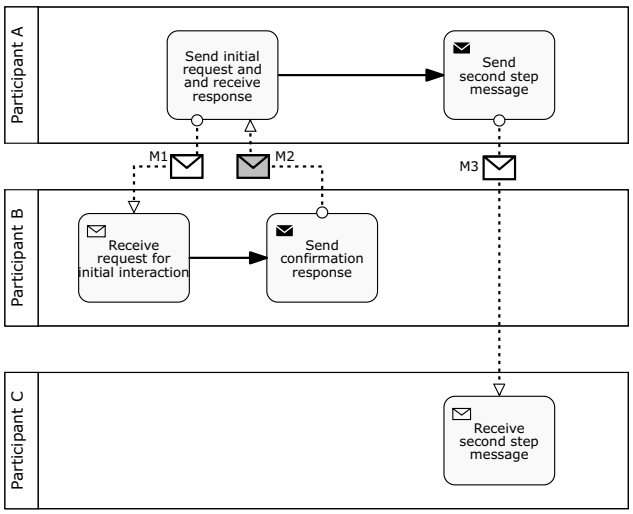
First we introduce the reader to a subset of BPMN terms and concepts, and refer to the standard [129] for an exhaustive description. Next, we characterize the particular aspects of WSN applications and how to model these aspects using the BPMN language.[1] Lastly, as an example, we model a WSN application for a parcel-monitoring scenario. Although we focus on BPMN as example notation for modeling business processes, the same concepts could be extended to other notations such as EPC [83] or UML [126] activity diagrams.

3.1 Background BPMN Terms and Concepts

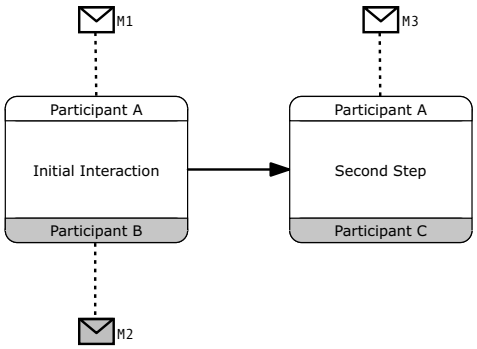
BPMN is a comprehensive graphical language which covers different aspects of business processes using different views as shown in Figure 3.1. All views use the same basic set of graphical symbols in a consistent way, which allows the diagrams to be easily interpreted by our targeted business audience.

The core of the BPMN language is represented by *collaboration diagrams* which describe in detail the behavior of entities involved in a business process as well as their communication interactions and the messages exchanged. The so-called *choreography diagrams* focus on the messages exchanged between entities, abstracting from the internal behavior of the entities involved. Choreography diagrams are complementary to collaboration diagrams and can help to disambiguate the ordering of message exchanges, thus removing possible deadlocks in communication. Finally, the coarsest view over a business process is a sketch of the communication interactions between entities, which is captured by

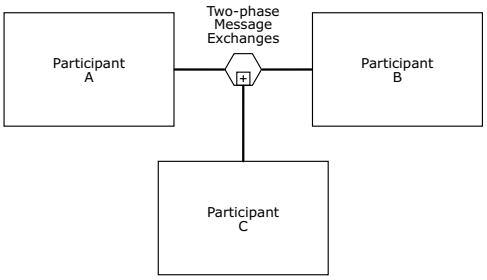
[1] A short version of this chapter has been published in [24].



(a) Collaboration



(b) Choreography



(c) Conversation

Figure 3.1: The three distinct views offered by BPMN for describing business process. The highest level of details is provided by (a) collaboration diagrams, less details are shown in (b) choreography diagrams, and finally (c) conversation diagrams only sketch the interactions in a process.

conversation diagrams. A conversation amasses a set of exchanged messages.

In our methodology, we focus on the BPMN collaboration diagrams because they can provide enough details which are required for execution. Throughout this work we will use the term model or process model interchangeably to refer to the BPMN collaboration diagrams. In addition, we use the conversation diagrams to provide additional, non-functional information such as network configuration and topology overview. We do not use choreography diagrams because they hide internal details related to data, events, and timings. Choreography diagrams are thus lacking the full information required for execution. However, they can provide a high-level overview for the behavior of a protocol in terms of exchanged messages. Thus, message traces could be automatically translated to choreography diagrams for analysis.

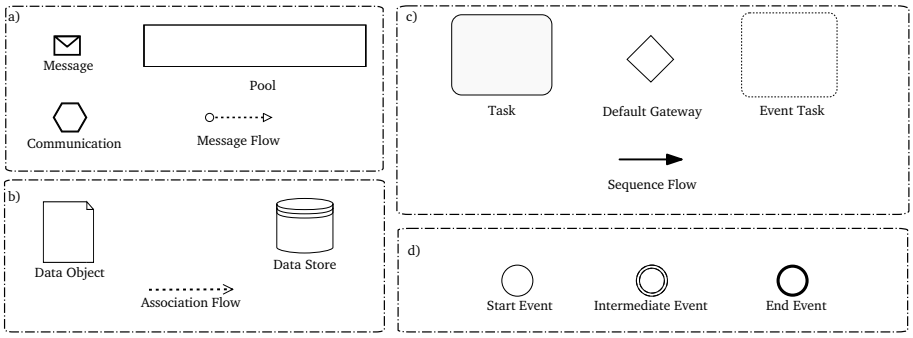


Figure 3.2: Summary of basic BPMN symbols for: a) communication flow, b) data flow, c) control flow and d) events. We use the BPMN *group* notation, rectangles with rounded corners and a dashed-dotted borders, to separate the different symbol types.

The BPMN graphical symbols can be classified in four main categories as summarized in Figure 3.2: a) communication flow, b) data flow, c) control flow, and d) events. The core set of BPMN symbols can be augmented by using visual aids, so-called *annotations*, which extend the semantics of components and are usually displayed in the lower part of graphical symbols. For example the plus-sign in square brackets ([+]) indicates that a component can be further refined into sub-components. This refinement method represents a powerful mechanism for encapsulating common and reusable functionality. Another often used annotation is represented by the parallel vertical lines (|||), which indicate that there are multiple instances for the respective component and that they are executing in parallel.

meta-model

All BPMN graphical components are based on a *meta-model* which describes the attributes of components and their relationships. For example, Figure 2.6(a) shows the meta-model for the `Message` graphical element. Only a subset of these meta-model attributes and data relationships are visible in the graphical representation. However, to allow a model to be fully executable, some of these *attributes* are required. They describe for example the end point for a service invocation, the reference to an operation, or the mapping of input and output parameters. The attributes are not represented graphically, but are accessible through a model editor and can be specified using tabular name/value forms.

These attributes refine the semantic details of a Level 2 operational model into a Level 3 executable model specification (cf. Figure 2.9).

3.1.1 Entities

Generally, a BPMN model describes a process with the interactions and behaviors of involved entities which cooperate in order to fulfill a required business goal. In BPMN terminology, a *pool* represents a participant entity and acts as the topmost container for its model. A pool may refer to a company, department, person, system, or, as in our case, to different sensor nodes.

The graphical representation of a pool is a rectangle which can be collapsed to visually compress a model view. When expanded the pool displays a detailed behavioral model as shown in Figure 3.1(a). The elements of the model are connected in a graph which can specify both the control flow, the data flow, as well as the communication flow.

3.1.2 Activities

The actual work performed by an entity is divided into *tasks* or activities which are shown using boxes with round corners. A basic *activity* for example is implemented as an invokable service or as a native operation for the target platform. To show the flow of control, i.e., that one activity follows another, *sequence flow* arrows are used.

A task may be recursively refined to contain other models in which case a plus-sign annotation is placed in the lower part of the box. Like the BPMN pool element, a sub-process can also be collapsed to provide a more compact visual representation, or expanded to show the full model details. Such an activity is then called a *sub-task* or *sub-process*. This encapsulation mechanism defines a hierarchy for the process model.

Details about the task invocation and references to services as well as inputs and outputs can be specified using the respective attributes. An additional visual aid can provide details about the task type and how this is performed, e.g., manual, user, service, script. Icons placed in the upper corner of a task box help distinguish among the different activity types.

A special class of activities are so-called *event sub-processes* or *event tasks* which are not part of the main control flow and are only started when the corresponding event occurs. As with normal sub-processes, these can be expanded and collapsed and contain further models. To distinguish such tasks from the ones that are part of the main control flow, the former are represented by a dotted border.

Event tasks cannot be connected to other components outside of their border using control flows. In a sense event tasks are akin to event handlers or interrupt routines typically used in embedded software. Event sub-processes contain tasks and events linked through control flow. Depending on the type of event which triggers the event task, the main control flow will continue to execute in parallel, or may be terminated. Moreover, the BPMN execution semantics allows for multiple event tasks to execute in parallel.

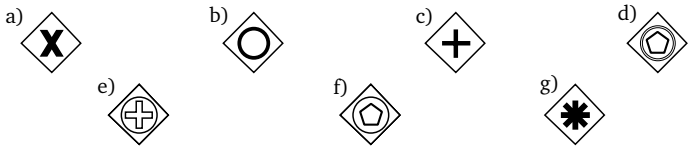


Figure 3.3: Summary of BPMN gateways for control flow: a) exclusive (XOR), b) inclusive (OR), c) parallel (AND), d) event-based, e) parallel event-based, f) exclusive event-based, and g) complex.

3.1.3 Gateways

Changes in the control flow are described using a *gateway* rhombus with an enclosed symbol which defines the semantics. A *parallel* (AND) gateway has a plus sign and forks the control flow if there are multiple outgoing sequences and joins the control flow if there are multiple incoming sequences. An *exclusive* (XOR) gateway has an X-symbol and represents a conditional decision point where the process follows the respective sequence flow that matches the condition. These gateways represent the basic constructs for controlling execution flow.


Apart from the basic gateways, BPMN provides additional ways to split and merge the control flow using more specialized gateways as shown in Figure 3.3. The *inclusive* (OR) gateway, depicted with a circle symbol, splits the control flow according to multiple, possible inclusive, conditions on the outgoing sequence flows and merges only sequence flows which have been activated.

Events are prime elements in BPMN models. As such there are several gateways which deal with how events are treated during the execution of a process. The *event-based* and *exclusive event-based* gateways both trigger only the sequence flow which corresponds to the first catch event outgoing from the gateway. They are both depicted using an enclosed pentagon with a thin border in the first case and a double border in the second case. The semantic difference between the two event gateways is that the latter can only be used to start a process, whereas the first can only be used as an intermediate element.


Another gateway used to start a process based on events is the *parallel event-based* gateway. To mark this particular semantic, the graphical depiction uses an encircled hollow plus sign. This gateway waits for the occurrence of all of the immediately following events.

Finally, the *complex* gateway does not have a clear semantic. For this gateway type, merging and forking can be described in natural language. Even though the complex gateway can greatly compact the graphical representation of a diagram, it is imprecise and not suitable for code transformation. Furthermore, the complex gateway type does not influence the expressibility of the language because different merging and forking patterns [170] can be expressed using the basic exclusive, inclusive, event-based, and/or parallel gateways. Consequently, we exclude the complex gateway from our methodology.

3.1.4 Data

The data objects which are required by a process to fulfill work can be explicitly represented in BPMN using *data items*, shown as papers folded on

the upper-right corner, and *data stores*, displayed as database cylinders. The difference between data items and data stores is that the former are volatile while the latter are persistent. Data items with a hollow arrow represent external input to a model, while data items with a filled arrow represent the output of the respective model.

Both are connected to readers and writers by means of *association flows* which are depicted as a dotted arrow. The direction of the arrow shows whether a read or a write operation is performed.

The visibility of data reflects the encapsulation mechanism of sub-processes. For example, a data object which is represented in a parent process is visible to all sub-processes, whereas a data object which is defined in a sub-process is not visible to the parent.

3.1.5 Communication

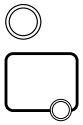
Communication is explicitly modeled using *communication flows* which are represented as dotted lines with a hollow arrow head on the receiver side and an empty circle at the sender side. The end points for communication can be either tasks, events or pools. However, message-based communication is only possible between different pools or entities.

An envelope on the communication flow arrow marks an actual definition for the communication. The envelope can include specific properties such as references to data or protocol information.

3.1.6 Events

Events are a key component in BPMN models. During execution, a process may wait for a certain *event* to occur, corresponding to a *catch* semantic, or it may trigger events, corresponding to a *throw* semantic, to notify different tasks or other pools. Examples of possible events include starting and ending a task, receiving or sending a message, and signaling exceptional conditions.

The BPMN graphical representation for events is a circle. To visually distinguish between the different event types and their properties, BPMN uses icons and different styles for the circle line as shown in Figure 3.4. Catch events have a hollow icon, while throw events have a filled icon. Furthermore, *start events* have a thin border and *end events* have a thick border.



Intermediate events which occur during the execution of a process but do not start or end the enclosing process are represented with a double border. Intermediate events can be part of sequence flows and can also be placed on the border of tasks. The semantic of a *border event* is to catch the respective event which is either thrown inside the task, or the external event which affects the respective task. A catch event that does not terminate the respective task or process has a dashed line. In contrast, catch events that do terminate the respective process or task are depicted with continuous lines.

In general, for a sound model for every event thrown there must be at least one corresponding event which catches this occurrence. There are certain exceptions to this rule. For example timer events are never thrown, they happen as normal execution and, conceptually, they can be interpreted as being thrown by the system or the environment as time elapses. An error event need not be thrown explicitly, it may occur due to internal state changes of a task and will

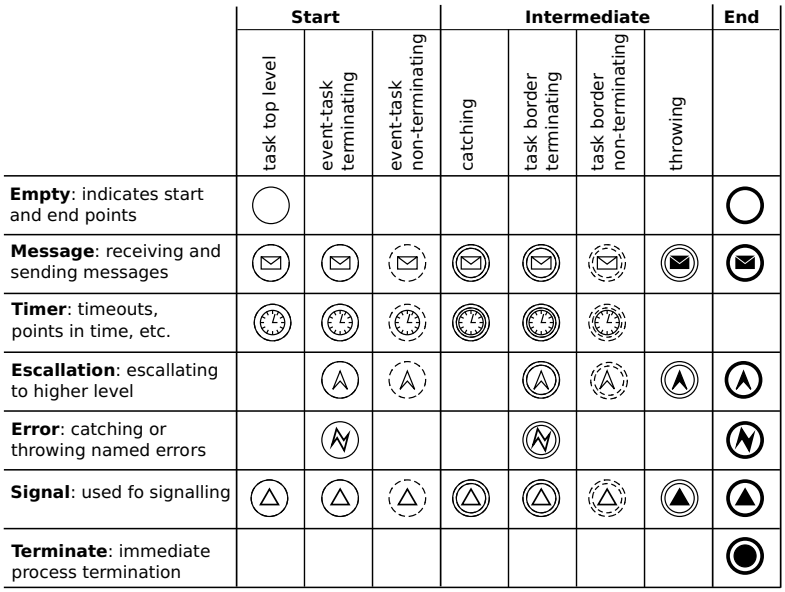


Figure 3.4: Summary of the different event types available in BPMN.

always terminate the respective task. Therefore the table in Figure 3.4 does not contain a non-interrupting catch event for the type error. The other exception are signal events, which represent a powerful and generic out-of-band broadcast mechanism. A thrown signal event can be caught by multiple catch events with the same signal identifier, across different entities and process hierarchies.

Figure 3.4 shows a subset of the event types defined by the BPMN standard which are sufficient for describing typical WSN applications. The BPMN includes further event types such as the *compensation events*, which are required for transaction support, or the *condition events*, which can be used to model complex rules. However, such event types are not common for WSN application. Moreover, it is possible to create equivalent models by replacing such events with specialized tasks and the more general signal and escalation events.

3.2 Mapping WSN to BPMN

WSN applications are characterized by their distributed nature, reactive behavior, unreliable wireless links, noisy sensors readings, real-time requirements, and usage of heterogeneous platforms. In the following, we define these core WSN aspects and show how they can be modeled using the BPMN language without extending or modifying its notation nor execution semantics.

A common architecture for WSN applications uses the following components: data acquisition and processing, network stack, gateway, visualization, and database. Thus, several tiers are involved to create a WSN application. The application executing on a mote is responsible for pre-processing any sensor data read and triggering actuators in response. The same or a different application can run a network protocol for wireless communication. Wired motes connected to a server represent the gateways to the wireless sensor network. A different

part of the application may run on the gateway which further processes and translates data, interprets commands, and resolves address. The complete application usually also includes some code to be executed on back-end servers performing data consolidation and forwarding, and a visualization component to properly display the gathered data.

3.2.1 Reactive-Behavior

Due to their reactive nature and for an efficient execution WSN applications rely on event-based run-time environments such as TinyOS [68], Contiki [37], or the IBM Mote Runner[22]. This behavior stems from the fact that wireless communication and sensor operations take a considerable amount of time. Actively polling for the result of actions will only spend energy while blocking the system from executing further operations.

In such cases, a more efficient implementation uses operations with an *asynchronous* interface which will trigger the action to be performed. If the system has no further actions to execute, a scheduler then decides to turn certain parts of the system off to save power, including the MCU which is put into an appropriate sleep mode. When the action completes, the system is asynchronously notified about the result using an event which is placed into an *event queue*. The scheduler monitors this event queue and passes the top event to the application for processing. The queue of events must not be necessarily implemented in software. It can for example directly use the interrupt vector provided by the MCU. In this way, multiple operations, such as radio communication and sensing, execute virtually in parallel and the run-time system can efficiently manage the scarce computational and energy resources.

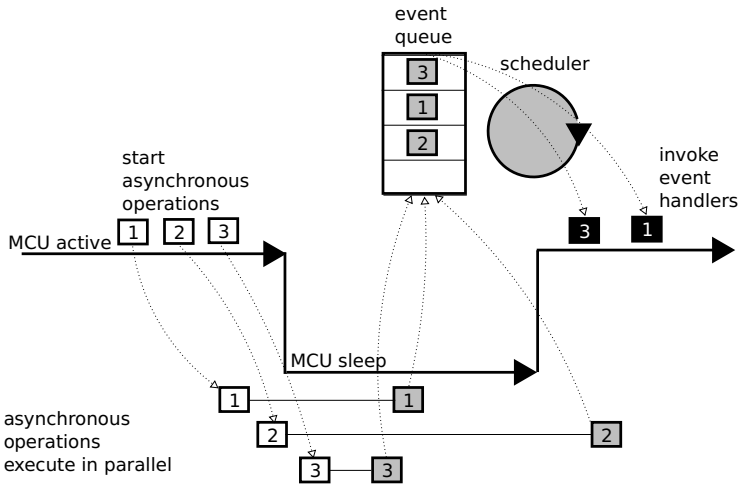


Figure 3.5: Illustration of asynchronous behavior and the scheduling of operations. The scheduler starts asynchronous activities, which may execute in parallel while the MCU is sleeping. After completion each activity enqueues an event. The queue is monitored by a scheduler, which invokes the corresponding handler for processing. In turn, further asynchronous operations may be started.

By contrast operations which execute to completion are *synchronous*. These are usually simple operations which can complete almost instantly such as enabling an LED, toggling a GPIO pin, or resetting the mote.

Abstract States

In general, an operation with an asynchronous interface can signal three distinct abstract states. The common scenario is the *normal completion* of the operation, preferably with one or more return values.

In addition, applications have to be prepared to deal with exception cases which deviate from the normal execution path. Such cases represent an *exception completion* of operations, preferably with the reason for the failure so that the application can react accordingly. For example, in a WSN, a hardware malfunction can occur while reading a sensor because the respective sensor board was detached. Consequently, the sampling operation fails, which means the respective activity is terminated with an exception.

Lastly, a *significant state* may occur while the asynchronous operation continues execution. As an example consider an application which continuously tracks beacon messages in a TDMA[2] communication protocol. In this case, an event notifies the application that a beacon was received, which represents a significant point for time synchronization. In contrast to the previous cases, the operation which tracks beacons remains active and can send further notifications to the application.

How to Model in BPMN?

The event construct provided by the BPMN language offers a rich palette for expressing such asynchronous behavior. In general, the matching between throw events and catch events is done primarily using their name and type. Further event attributes can be used to disambiguate multiple matches. Whereas, the name and type are displayed graphically, the attributes which refine the event type are only accessible in a tabular form. For example, an escalation event has an `escalationCode` attribute which specifies a string value. In this case, throw events will only match catch events with the same escalation code. In addition, events can carry data and thus have references to data objects.

event matching

As a general rule, events are always propagated upwards in the hierarchy provided by the BPMN encapsulation mechanism. That is events travel from a sub-process to the enclosing process. Moreover, events are only visible inside the pool which represent the entity where they occur. In other words, events in one pool cannot influence events in other pools. Once caught an event is immediately consumed. Thus, an event cannot be caught in two different places.

An exception to the above rules is represented by the BPMN signal events which propagate in both hierarchical directions and across pool boundaries and they can be caught multiple times. This type of event represents a very powerful out-of-band communication mechanism.

In most cases the matching of events can be decided statically. However, some cases depend on the current state of the execution. In this latter case, matching can thus only be decided at run-time. For the simplest static analysis,

[2]Time Division Multiple Access (TDMA) is a channel access method for shared medium networks.

throw events for which there exist no corresponding catch event in the process model are simply discarded. This case can be considered as an incomplete model and the modeler should be warned. If the process model contains a matching catch event for a corresponding throw event we have to distinguish between two cases.

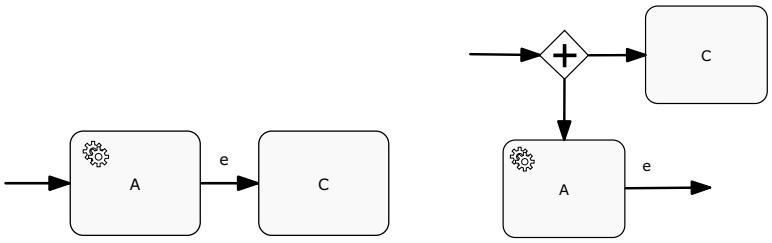
event activation

The first case corresponds to an active catch event. We define a catch event to be *active* if in the current execution state, a sequence flow reached the catch event. For catch events which start a task or event-task this condition corresponds to the enclosing task being active. In this case, the catch event immediately consumes the thrown event, i.e., execution continues on the outgoing sequence flow from the catch event. Once the event is consumed, it cannot be caught by another catch event.

In the second case the corresponding catch event is inactive, i.e., in the current execution state no sequence flow reached the catch event. For catch events which start a task or event-task this corresponds to the enclosing task being inactive. In this case, the event occurrence is discarded.

normal completion

To describe asynchronous behavior with BPMN, we first consider the events which signal the *normal completion* of an asynchronous operation. Because the normal completion of a asynchronous operation is unique, we model this implicitly using an outgoing sequence flow, as shown in Figure 3.6(a). When an execution token reaches `A` the task becomes active, which in our case corresponds to starting the asynchronous operation. The sequence flow `e` is only activated when the task `A` completed normally, in our case the asynchronous operation signals its completion. Only afterward, the task `C` becomes active and execution continues. This behavior is according to the BPMN execution semantics.



(a) (b)

Figure 3.6: (a) Normal completion of an asynchronous operation. (b) Starting an asynchronous operation in parallel.

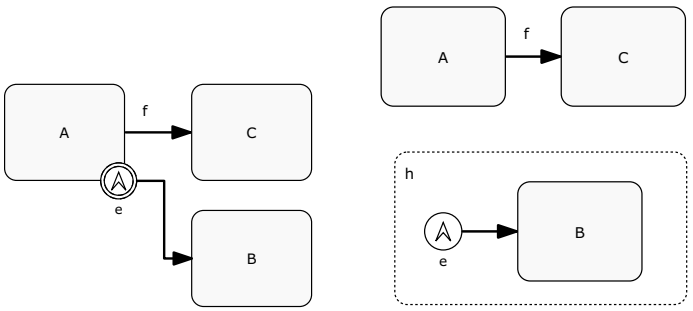
The BPMN language per-se does not define asynchronous operations, this is a notion we introduced in the context of WSN applications. As such, there is no special annotation which can be used for this type of task. However, the BPMN language allows the definition of additional task types. Such an extension would help the visual distinction between tasks which execute to completion and asynchronous tasks. Consider that instead of task `C` executing only after task `A` signaled completion we wish to start `C` as soon as `A` is started. This case can be expressed using a parallel gateway as shown in Figure 3.6(b). If the control flow is split, the modeler must take care that merging is specified in a

consistent manner. If the parallel gateway is omitted, the outgoing operations following sequence flow `e` will be executed only after the asynchronous operation completes.

Note that in Figure 3.6, as an example, we annotated the asynchronous operation `A` using the BPMN `service` symbol represented by the two cog-wheels. Such an annotation for asynchronous tasks offers an explicit visual distinction from basic operations.

exception completion

In contrast to the normal completion events, the *exception completion* of a asynchronous operation can only be captured using interrupting border events. For conditions which occur internal to the operation, we choose the BPMN escalation event type as this type of event can only be thrown from within the respective task or process. Other types of events which can notify about exception condition which are timeouts, messages, signals, and cancellations. These are represented using the respective BPMN event symbols (cf. Figure 3.4). The main difference is that these events are triggered by factors external to the activity on whose border they are attached. Figure 3.7(a) shows a task `A` which is interrupted as soon as the event `e` is consumed by the catch event. The event is caught on the border and execution continues with task `B`. For critical exceptions which are triggered by the execution of the operation, we employ the BPMN error event symbol.



(a) (b)

Figure 3.7: Exception completion of (a) a task or (b) the entire process.

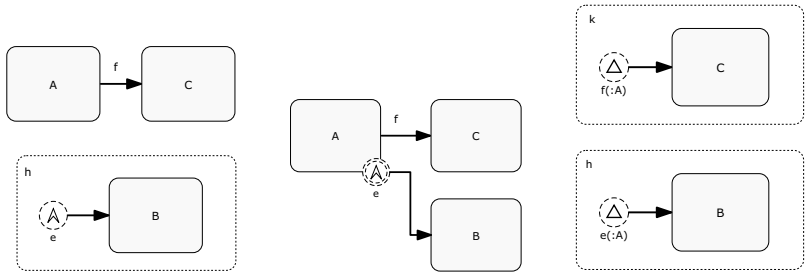
In addition, BPMN allows to model a stronger exception which non only signals the interruption of a particular task but of the entire enclosing task. Such an event could for example signal a watchdog timer interrupt or a system critical error, e.g., memory access violation. Such exceptions can be modelled using event-tasks with a start event with an interrupting semantic. In Figure 3.7(b), the task `A` can trigger the event `e`. If this situation occurs, then the handler `h` catches the event and the enclosing process is terminated. Subsequently, `C` is executed, whereas `B` is never executed.

significant state

For describing the occurrence of a significant state during the execution of the process the BPMN language provides several possibilities using non-interrupting events. The first approach, shown in Figure 3.8(a), explicitly uses the task `A` to mark the start of the respective asynchronous operation. At a later point this task will throw an event `e` which is handled by the event-task `h`. The

handler then executes a task `B`. Note that `h` does not terminate the enclosing task because the process should be able to continue handling other events. This type of modeling is the closest to a typical event-based program. However, such a modeling style might be hard to manage and understand without a visual aid as it grows in complexity with the number of events. As an aid, the model editor could highlight possible throw events when corresponding catch events are selected and vice-versa.

Another form of modeling the occurrence of significant states is by using non-interrupting border events attached to tasks. Figure 3.8(b) is semantically equivalent with the previous example. In contrast to the border events in Figure 3.7(a), the event `e` is non-interrupting which is represented by the dotted lines. Also note that a sequence flow entering task `C` is only activated once the task `A` completed normally. As long as `A` is active it can potentially generate multiple events of type `e`. All these events are processed without terminating task `A` and for each event `e` task `B` is executed. The advantage of using this approach is that tasks which are asynchronous and corresponding events are visually linked and the start and end of such activities is explicit. We favor this approach in particular user-defined task and sub-processes.



(a) (b) (c)

Figure 3.8: Different ways of modeling significant state for reactive asynchronous operations. We prefer variant (b) because the binding between activities and events is explicit and clearly visible in the graphical model.

A modeling variation which uses several implicit assumptions is shown in Figure 3.8(c). The first assumption is that the activity `A` should be started automatically as soon as the enclosing process is active. The second assumption is that the event `e` can only be triggered by activity `A`. To maintain this causal relation we use the name `e(:A)` for the start event which triggers the handler `h`. The main difference to the previous two variants is that in this case `e` can occur as soon as the enclosing process in active, whereas in the previous variants, `e` can only occur once task `A` is active. The same implicit assumptions could be applied for the event `f(:A)` which notifies about the normal completion of activity `A`. The advantage of this approach is that it provides a good visual separation for event handlers. Moreover, if activity `A` never completes, the event handler `k` can be omitted, which results in a visually compact representation. The disadvantage is that this modeling style uses implicit assumptions which must be understood by the modeler and the control flow inside the event handlers cannot

be merged with control flow outside the respective event-tasks. For frequently used activities such as a watchdog timer, or an always active debug interface, this solution is well suited. When control flow clarity is more important than visual compactness, the two previous styles in Figure 3.8 (a) and (b) represent more general solutions.

In practice, the semantic of events generated by asynchronous operations is defined in the documentation for the respective platform API. Alone from the corresponding function signature or even provided with the source code, it is not possible to infer whether an event signals normal or exceptional completion of an activity. Moreover, it is not possible to distinguish whether the event simply notifies about a significant state while the operation continues its activity. In absence of source-code annotations for the asynchronous operations provided by the platform, translating an API to BPMN symbols is thus semi-automatic. With appropriate API annotations, this translation can be fully automated, and for each API function the corresponding BPMN task with the appropriate border events can be generated. These annotations can be provided by developers at the same time as writing the documentation comments.

3.2.2 Heterogeneous

In general, a WSN is composed of several heterogeneous nodes with different roles in the network. The hardware capabilities define the role a mote can take in a WSN. In terms of hardware, there is a large choice available for MCUs, radio chips, power supplies, serial communication peripherals, as well as highly specialized sensors and actuators. These components represent the lowest layer for the hardware platform.

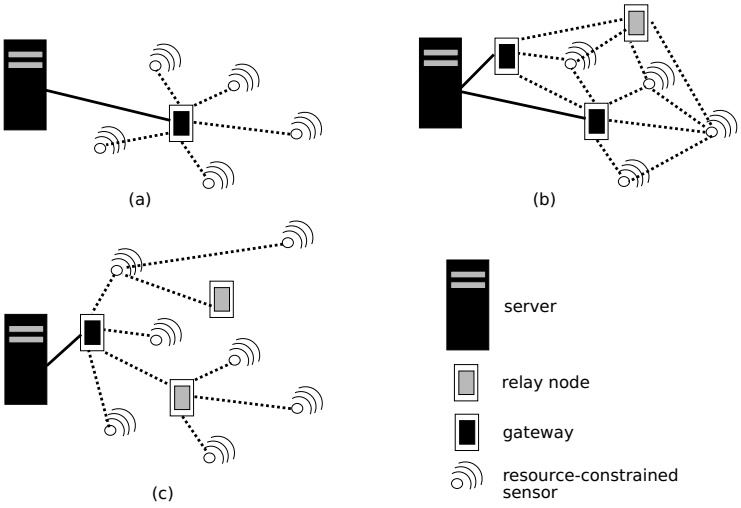


Figure 3.9: Illustration of different topologies typically used by WSN applications: (a) single star, (b) mesh, and (c) tree. Sensors, relay nodes, and gateways communicate wirelessly, whereas servers and gateways are wired.

For example, a constantly powered gateway node can serve as the entry point into the WSN forwarding and possibly translating messages to back-end

servers from the WSN and vice-versa. Other motes which are equipped with a more powerful MCU and a larger battery supply can serve as relay points for network traffic. Finally, the cheaper and resource-constrained devices operating on batteries represent the leaf nodes in the WSN which sample, process and forward sensor information to the gateway.

To abstract from the hardware details a general approach is to introduce abstraction layers. Figure 3.10 shows a typical layered architecture for a sensor node. The first such layer is the hardware abstraction layer (HAL) which hides the details of the different MCUs, radio controllers and peripherals. For example, the HAL unifies the functionality provided by MCU's hardware timers using portable time units such as seconds and milliseconds, allows to reset the mote, or operate simple devices such as LEDs. On top of this layer the OS manages the resources of the node and the scheduling of tasks while controlling power consumption. In addition, the OS may provide built-in communication protocols, application management, and debugging support. Ideally, the OS should expose the hardware sensors and actuators using portable units, e.g., temperature in °C, acceleration in m/s^2, or light intensity in lux.

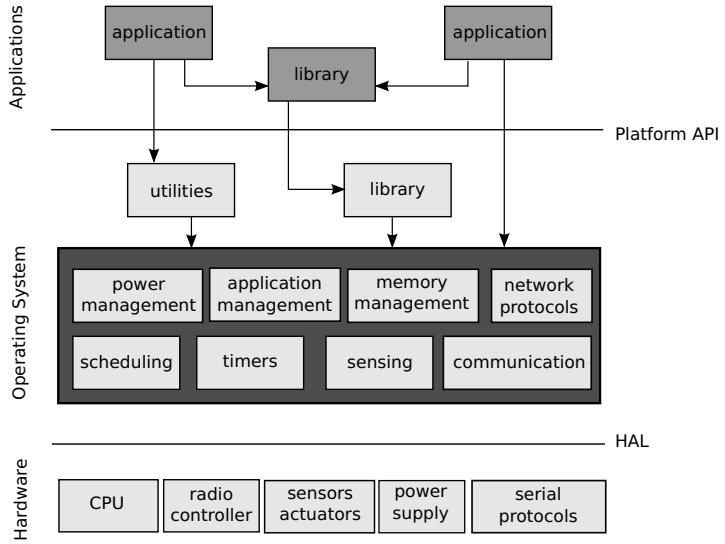


Figure 3.10: Software layers for the platform API which abstract from the underlying hardware.

platform API

Programmers typically use a *platform API* to develop WSN applications. This API represents the interface between applications and the OS as well as additional utility libraries which are part of the portfolio of a particular development environment. The platform API is defined by both the hardware as well as the OS choice. For example for TinyOS, the platform API would include libraries for communication protocols such as the default collection tree protocol [55].

Whether certain interfaces are available for a platform API depends on the underlying hardware. In particular, in term of wireless communication, the radio chip defines what physical layer protocols are available. For example, a

mote which only has a BlueTooth chip cannot use ZigBee for communication. Moreover, the platform API may add common abstractions and, where needed, even the functionality which is not implemented directly in hardware.[3] Whether a certain API functionality is implemented or available on a particular hardware can be detected by the application either at compile time, as it is the case for static libraries in TinyOS, or at run-time using dynamic libraries, for example in Mote Runner.

How to Model in BPMN?

To bind a certain entity in a model to a particular WSN platform the properties of BPMN pool can be used, in particular the *processCategory*. Based on this selection the corresponding model editor could display only the BPMN tasks which are available for the corresponding platform API. An alternative is to specify the platform API only when models are compiled to executable applications. This approach will lead to errors when libraries which must be resolved statically are not available for the corresponding platform.

However, for portable business processes which allow migration to improved hardware platforms this approach should be avoided. Instead, the execution platform should be flexible and allow for execution of the same application model independent of the actual hardware platform. This criterion is fulfilled by using a portable VM [72] which abstracts from the underlying hardware.

3.2.3 Communication

A distributed application spreads the execution of work among participants which are required to cooperate using communication primitives. In general, communication has three main aspects: the *protocol* used to exchange messages, the *format* for each message, and the *unique addresses* which distinguish between different entities such that messages are delivered to the required destination. Furthermore, the physical *topology* of a WSN defines how the sensor nodes are distributed in the environment. The topology can be fixed or can change dynamically based on environmental and mobility conditions of sensor nodes.

protocol

Literature provides a vast array of custom WSN communication protocols [7] as well as different standards concerned with the the different communication layers in the (Open Systems Interconnection) OSI [189] reference architecture in a WSN. Standard examples include the IEEE 802.15.4 [77] for the physical and data link layers, the 6LoWPAN [153] for the network layer, and the various application profiles for ZigBee [188].

addressing

For address resolution, communication requires each sensor to have a unique address. Typically in WSN each node has a globally unique 64-bit address, the so-called *extended unique identifier* (EUI), encoded in the hardware at manufacturing. This is akin to the medium access control (MAC) address used by network interface cards in standard Internet protocol (IP) networks.

abstractions

To deal with the complexity of describing and programming distributed WSN applications by specifying the behavior of individual nodes, several abstractions

[3] At the time of writing, the slotted Carrier Sense Multiple Access (CSMA) for the IEEE 802.15.4 MAC protocol, for example, is not implemented in hardware by most modern radio chips.

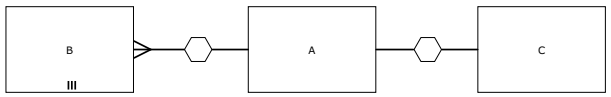
have been proposed also referred to as *macro-programming* [114]. These abstractions allow to view groups of nodes or the entire WSN as a single logical entity. Addressing multiple nodes is achieved using the group abstraction which can be based on logical properties or the physical topology.

From a technical perspective in a WSN the entities of interest are the sensor nodes themselves. From a business perspective the *entity of interest*, as defined in [63], has some attributes to be monitored and controlled by sensor devices. For example, a fridge could be an entity of interest which encompasses several sensor devices to control the temperature inside the fridge, the internal light, and the door, and possible a central control device which manages the contents of the fridge. The control device also represents the central access point for communication to and from the entity of interest, in this case, the fridge.

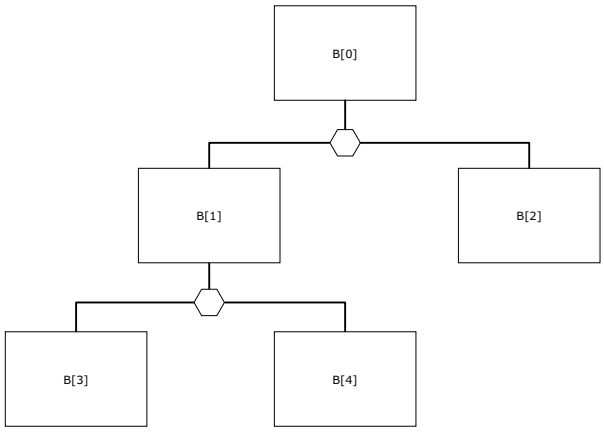
How to Model in BPMN?

topology

To describe the network configuration and topology we use the BPMN conversation diagrams. A network configuration is a sketch of how many sensor nodes are part of the network, what is their behavior type and logical connectivity. We represent groups of nodes in BPMN using parallel multi-instance pools. In this case, the groups are defined based on common behavior.



(a)



(b)

Figure 3.11: (a) Network overview and (b) fixed tree topology showing the logical connectivity.

Figure 3.11(a) shows a network configuration with three types of entities. There is a only one instance for both the entities A and C, whereas there are multiple instances of type B running in parallel, forming a logical group. The

cardinality for a certain entity type can be set using the *participantMultiplicity* property, whereas the choice for a platform API can be specified in the *processCategory* property. These attributes are part of the standard BPMN meta-model for the pool data type.

Figure 3.11(b) shows a static tree topology for the entities of group B with five nodes indexed starting at zero. If the network topology is dynamic the same notation can be used. In the later case, the diagram represents a snapshot of the network configuration at a specific point in time.

configuration

The network configuration in a simulation environment might also include parameters which characterize the real environment such as packet error rate, sensor noise, clock drift, the physical position in a given coordinate system, or the node failure probability. These properties are missing from the above diagrams but are required for simulations and testing which closely match real deployments.

There are two ways to integrate this information. The first approach uses the pool name to append the additional information. As an example, consider the following text-based encoding:

```
B[2] @ (0,0,0)
```

which can specify that the mote of type B with index 2 should be positioned at the coordinates $x = 0$, $y = 0$, and $z = 0$. This approach requires a language with a particular syntax to specify all required properties. The disadvantage is that the pool names might quickly become unreadable.

The second option is to extend the attributes of the BPMN meta-model for the pool class and allowing these additional attributes to be edited in a tabular form. Fortunately, the BPMN meta-model was designed with such extensions in mind. For this purpose, the `extensionDefinitions` and `extensionValues` attributes of the BPMN `BaseElement` are used. Note that in our case, these extensions do not modify the execution semantics nor syntax of the BPMN language, they are simply aids in defining a detailed network configuration. The second method is more robust and leaves no room for interpretation or syntax errors.

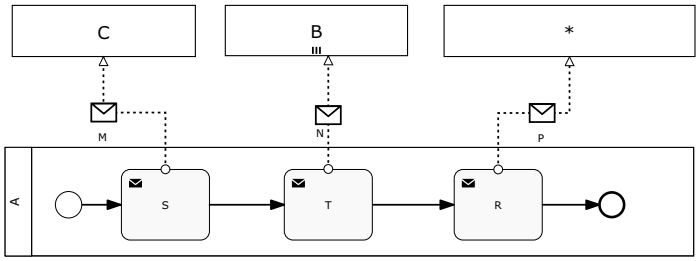
addressing

In a BPMN process, communication is modeled explicitly with sender and receiver bound to the start and end of message flows respectively. Thus, as a general address resolution principle, entities are identified by their name. We distinguish between two cases for addressing: single nodes and groups of nodes.

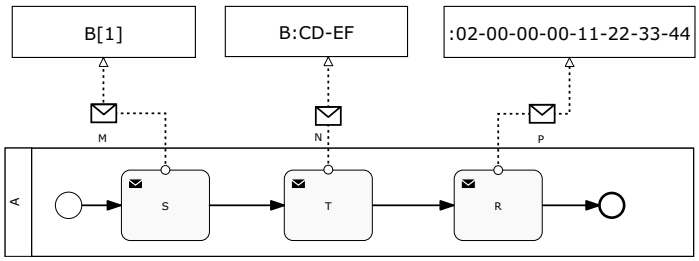
In the case where the entity name describes a single instance entity the mapping to an address is straightforward. This case is a particular instance of a group with a single element. An example of this case is the message `M` in Figure 3.12(a) which is sent by task `S` from entity `A` and is addressed to the single instance entity `C`. In case the entity represents multiple instances, the message is delivered to all entities in the respective group.[4] An example of this case is the message `N` in Figure 3.12(a) which is sent by task `T` from entity `A` and is addressed to all instances of type `B`.

Furthermore, to support wireless communication protocols which use broadcast messages, we define a special group for which we use the `*` wildcard. This group refers to a broadcast message which is received by all direct neighbors of the sender with access to the physical medium. This case is exemplified by the

[4]This functionality must be supplied by the corresponding communication protocol.



(a)



(b)

Figure 3.12: Addressing (a) groups of motes and (b) unique motes.

message `P` in Figure 3.12(a) which is sent by task `R` from entity `A` and is addressed to all one-hop physical neighbors, irrespective of their type. Depending on their physical location, both `C` and possibly all nodes of type `B` will receive the message.

To address a particular node in a group we use either the indexing as defined by the fixed topology, the unique 64-bit, or the short 16-bit address.[5] For example, in Figure 3.12(b), the message `M` is sent by a task `S` from entity `A` is addressed to an entity instance of type `B` with index `1` in the topology. A different method of specifying the destination is used in Figure 3.12(b). Th message `N` is sent by task `T` from entity `A` is addressed to an entity instance of type `B` with the short address `CD-EF`.

Often the address for the communication partner is not known and can change dynamically according to changes in the network topology during the process execution. Consequently, a variable data item can be used to specify the group index, the short address, or the full address of an entity.

message exchange protocol

The communication protocol can be specified as a property of the message envelope. For this purpose we use the BPMN `category` attribute. This property can either refer to an existing standard WSN protocol such as ZigBee or 6LoWPAN, a well established communication protocol such as UDP, TCP/IP for wired communication, or simply the raw frame format of the IEEE 802.15.4 wireless data links. Note that, the choice of protocols is limited by the available implementations for the specified execution platform as defined in the enclosing pool. The building blocks for sending and receiving messages using a specific

[5] According to the IEEE 802.15.4 specification.

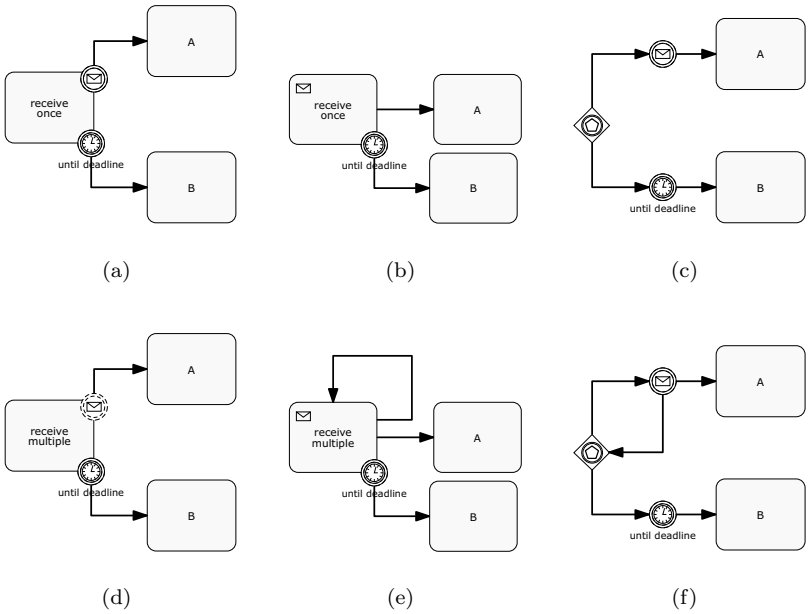


Figure 3.13: Different possibilities of modeling for receiving one single message (a,b,c) or possibly multiple messages (e,f,g) with a timeout deadline. The `receive once` and `receive multiple` tasks are provided as building blocks by the respective platform API.

communication protocol would appear as send and receive building blocks when refining a pool. To visually distinguish such communication tasks from other activities in the model, we use annotations with corresponding receive and send envelopes as depicted in Figure 3.13.

For a consistent model, the communication protocol specified in the message envelope must match the end points which are bound to both the sender and the receiver pools or respective tasks. As a default case, when no protocol is specified by the message envelope nor the respective send and receive tasks or events, communication is mapped to the best available protocol in the OSI layer. In the case of a typical WSN network, messages without a specified protocol are implicitly mapped to the IEEE 802.15.4 physical frame format.

message format
header & payload

Furthermore, the *header* of the message is automatically initialized based on the address resolution scheme described previously and in accordance with the specified protocol. For example, when using the raw IEEE 802.15.4 format, we define for each group of sensors a personal area network identifier (PANID) based on the hash of the group name. Messages which are addressed to all nodes in a group will use the `BEACON` frame type, the PANID of the group, and a broadcast short destination address (`DST=0xFFFF`). Whereas messages which are addressed to all physical neighbors, i.e., the previously introduced * group, will use both the broadcast destination address and the broadcast `PANID=0xFFFF`. Furthermore, for messages which are addressed to a single destination we use the `DATA` frame type with source and destination addressing and requesting acknowledgements.

Modeling communication is thus flexible and exchanging the protocol only involves adapting the message envelopes and the respective tasks. As this approach is not dependent on a particular communication protocol, other standards can be easily supported such as Web Services, provided a corresponding (and efficient) implementation exists for the respective execution platform.

Another function of a BPMN message envelope is a container for the data *payload* which is transferred between entities. The data structure for the payload can be specified directly in the BPMN model or by using a reference to an existing, typically more complex, model from an object repository. The data models in the repository can be defined using other modeling tools such as the UML class diagrams. To define the data structure directly in the BPMN model itself we use the *ItemDefinition* property which allows to specify the name, data type, and optionally a state or value.

To transfer the payload the corresponding data structure must be serialized by the sending activity and de-serialized by the corresponding receive activity. Using association flows on the sender side data objects are pushed into the send activity, and on the receiver side the data objects are extracted from the corresponding receive activity. We use the name of the data objects for binding to the correct items in the payload. Items in the payload which are not specified will simply have a default value according to the initialization of the message buffer.

further considerations

The automatic mapping from high-level BPMN communication tasks to the underlying protocol on the execution platform hides certain parameters. For efficiency reasons, e.g., packing even more data into messages, certain fields could be left out. For example, data frames without source address specification may provide 4 additional bytes which can be used by the payload. To allow such detailed communication properties to be minutely controlled in the model, the message envelope can be used to specify the exact raw data bytes which should be transferred instead. This specification must then include the header information which specifies the addressing information and the data frame type. However, such a specification represents a very fine level of detail, which makes models less portable and very technical. Consequently, such details should be encoded in communication building blocks which can be reused by models with guaranteed efficiency.

The natural way to represent an entity of interest under one pool is by using the BPMN concept of *swim lanes*.[6] In this case communication among swim lanes must occur implicitly through the sequence flow of tasks, shared data objects, or more generally signal events. This restriction is imposed by the BPMN standard which specifies that message flows must not be used inside the same pool. This definition contradicts the natural fact that the sensors pertaining to an entity of interest use wireless communication for collaboration. In this case, only communication to and from the entity of interest can use messages flows and envelopes.

A more viable solution is to model the entity of interest and the enclosing sensors using individual pools. Subsequently, use the BPMN *group* notation to mark the inclusion relationship between the respective sensors and the entity of interest. We favor this approach because it imposes no restrictions on model-

[6] A swim lane can divide the activities in a pool into functional areas. For example, in a company a swim lane may correspond directly to a department or team (cf. Figure 2.1).

ing the message-based communication between devices pertaining to the logical entity of interest.

3.2.4 Real Time

Communication protocols and sensing in a WSN are minutely scheduled for efficiency reasons and to meet the sampling requirements of the application. Thus, the notion of an accurate real-time is essential for the correct and efficient functionality of such a network. For example when using a TDMA protocol for communication all devices must wake-up and receive a beacon in a synchronized manner. Subsequently, nodes then perform useful work in their allocated time slot, and forward their local results towards the gateway nodes, possibly via peer or relay nodes.

global time vs. local time

Each mote in a network can have two views over the passing of time: a global view and a local view. Global time is based on the universal time, e.g., UTC from GPS, and is linked to different calendars. Local time starts when the mote is powered up and can in general be derived from the global time. However, for typical WSN applications, it suffices for individual sensor nodes to keep track of an accurate local time and occasionally synchronize with a peer which has an accurate view of the global time. In this case, motes require a common understanding of *time intervals*.

To keep the network synchronized, the drift in the local clocks should be minimized. For example, the IEEE 802.15.4 standard specifies a maxim drift of 20 ppm which ensures correct functionality of the communication protocol and higher layers including ZigBee and 6LoPAN. Different time synchronization protocols such as [98, 145, 158] can be included as part of the OS or as library. Alternatively, for open-air environments, some nodes in the network could be equipped with GPS modules which supplies an accurate global time.

How to Model in BPMN?

Consequently, a modeling language must have the ability to specify time, both global and local. In BPMN this is possible using timer events. For this purpose the mutually exclusive BPMN properties `timeDate`, `timeCycle`, `timeDuration` allow specifying the date and time for timer events using the ISO-8601 format for representing date and time, recurring time intervals, and non-recurring time-intervals. These specifications are all from the perspective of a global time.

For WSN, we focus on the local time as viewed by a sensor node as this perspective is sufficient to specify a mote's behavior. We thus assume that the platform API provides means to query the current local time. The translations to and from a global time or calendar dates can also be provided as utility functions by the platform API.

time predicates

To provide a quick overview, we suggest using the name of the timer events to specify time predicates using the syntax

```
PREPOSITION TIME [UNIT]
```
(3.1)

Furthermore, in the local frame reference of the mote, we allow for both relative and absolute time specifications. The preposition `at` is used for precise moments in time, whereas the preposition `in` defines time spans starting from the moment a sequence flow token reaches the timer event. The time value can be either a

constant number or the name of a variable. The units we support are s, ms, and platform specific ticks which depending on platform are roughly equivalent to μs. If the unit is omitted, the most accurate unit is assumed, in our case these are the platform specific ticks.

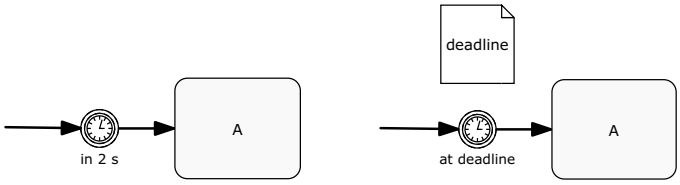


Figure 3.14: Examples of time specifications.

Figure 3.14 shows two examples which specify in the mote's reference frame a) a relative time using a constant unit, and b) an absolute time using a data item. In both case, as soon as the respective point in time is reached activity `A` is executed.

3.2.5 Synchronization and Data Races

BPMN processes represent a parallel execution model. Because of this, with respect to synchronized access to data, they exhibit the same set of problems as parallel programs in multi-threaded environments. In a parallel program, a *data race* occurs when multiple threads of execution access the same logical data without synchronization and at least one of the accesses is a write operation. To guard against such situations, there are well-known mechanisms. These ensure access to data is synchronized based on assumptions which are either optimistic [86] such as transactions, or pessimistic such as locking [40].

Even though BPMN provides a transaction task type and methods for compensating the effects of a failed transaction, the language lacks explicit constructs for synchronized data access. These aspects are discussed by [150] which proposes extending the language with data synchronization mechanisms such as locking.

Instead, this thesis aims to avoid language extensions. Hence, we propose using tools for analysis which provide warnings when data races may occur together with possible solutions. The suggested solutions could for example serialize the parallel execution if the users decides parallel execution is not required on a case by case basis.

The model in Figure 3.15(a) shows a data race as both tasks `A` and `B` execute in parallel and they both read and write to the shared data item `d`. This situation cannot be disambiguated without data access synchronization mechanisms such as locking or transactions to ensure a consistent state for the data item. However such synchronization mechanisms are expensive and often data race situations represent faults in the model or the requirements of the application. A possible solution is to revisit the application specifications and modify the model such

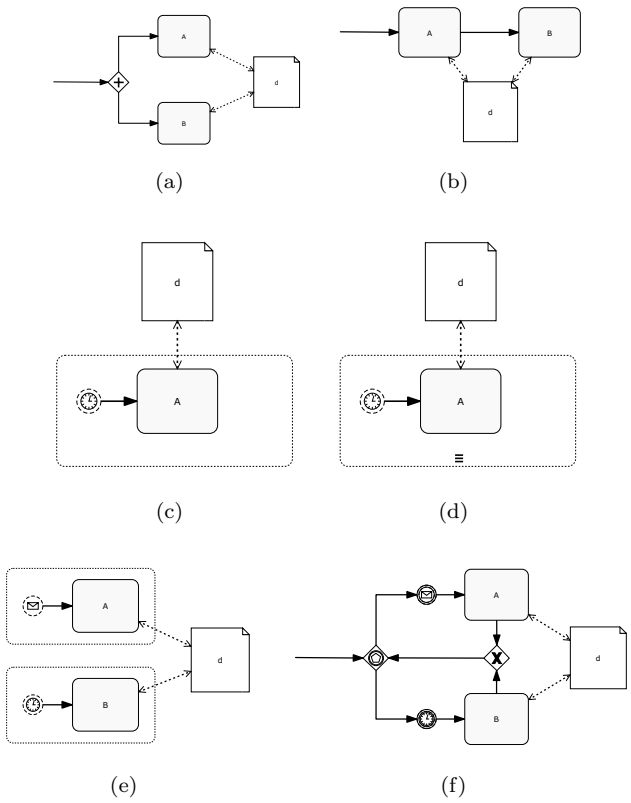


Figure 3.15: Race conditions (a, c, e) and disambiguating solutions (b, d, f).

that the two tasks are executed sequentially which ensures a deterministic order as shown in Figure 3.15(b).

There are other, non-trivial situations where data races can occur in BPMN models. One such case occurs when using a single non-interrupting event task as shown in Figure 3.15(c). In our example the event-task is started by a non-interrupting timer event. Because the event-task becomes active as soon as the respective event occurs and assuming the event can be triggered multiple times in parallel, we again have a data race. In this case, multiple instances of the task `A` will read and write to shared data without synchronization. This case can be disambiguated by explicitly annotating the event task with a *multiple instance sequential* (≡) behavior as shown in Figure 3.15(d). If such as annotation is present, the parallel execution of multiple event-tasks is serialized.

Another data race situation occurs when two different event tasks perform read/write access on the same data item as shown in Figure 3.15(e). This situation cannot be disambiguated by the previously used ≡ annotation because it applies only to the corresponding event task. In our case, multiple instances triggered by the message event will be sequentialized. However, the timer and message events can occur independently and in parallel. Figure 3.15(f) shows a possible solution which explicitly models a dispatcher by using an event-based gateway. The gateway only activates the sequence flow of the first occurring

event, i.e., either task `A` or task `B` will access `D` depending on whether the message or the timer event occurred first. This solution assumes that events are queued until they are processed, and we loop back to the event-based gateway to process the next event in this queue.

Several algorithms for detecting data races in parallel programs have been proposed [82, 127, 187]. For example, the Nondeterminator-3 [82] algorithm itself executes in parallel. Whereas, the HARD [187] algorithm provides a hardware implementation. Such algorithms offer useful debugging and analysis support which, if integrated in the modeling tools, can warn the user and suggest possible resolutions similar to the previous examples.

In general, WSN platforms only have one processing unit and use either an event-based execution model or a cooperative thread-based model. This means that synchronous tasks will always execute to completion before a new task can start execution. Thus, data races between synchronous tasks cannot occur. However, as soon as asynchronous operations are involved data races can occur. Consequently, synchronous tasks can be used as global locks for synchronization.

3.3 Example Use Case

To show how the presented BPMN modeling style can be used to describe WSN application in practice we use a typical application scenario.[7] We consider a business process which describes the delivery of parcels containing sensitive goods. The process ensures that the environmental temperature stays within allowed thresholds inside the parcel to meet the quality requirements for the delivered goods. The process involves the headquarters, a truck, parcels and goods. The truck ships parcels from the headquarters to a destination and each parcel contains goods and a WSN node which monitors the temperature. In the truck there is a base node equipped with mobile Internet connectivity which communicates with both the parcels and the headquarters. This business process is intended to be part of a delivery company's overall operations and is executed for each set of goods which has to be delivered to a specific destination by a single truck.

Figure 3.16: WSN topology for the parcel delivery process.

Figure 3.16 shows the topology for the WSN using a BPMN conversation diagram. The `Parcel` sensor nodes are directly connected to the `Truck` using a star topology. The `Truck` represents the gateway between the wireless sensors in parcels and the `Headquarters`. This diagram represents the top-most level in the model hierarchy.

Additional details can be added to this conversation model. For example, Figure 3.17 shows all the systems and entities involved in the entire parcel

[7] A shorter version of this example has been published in [21].

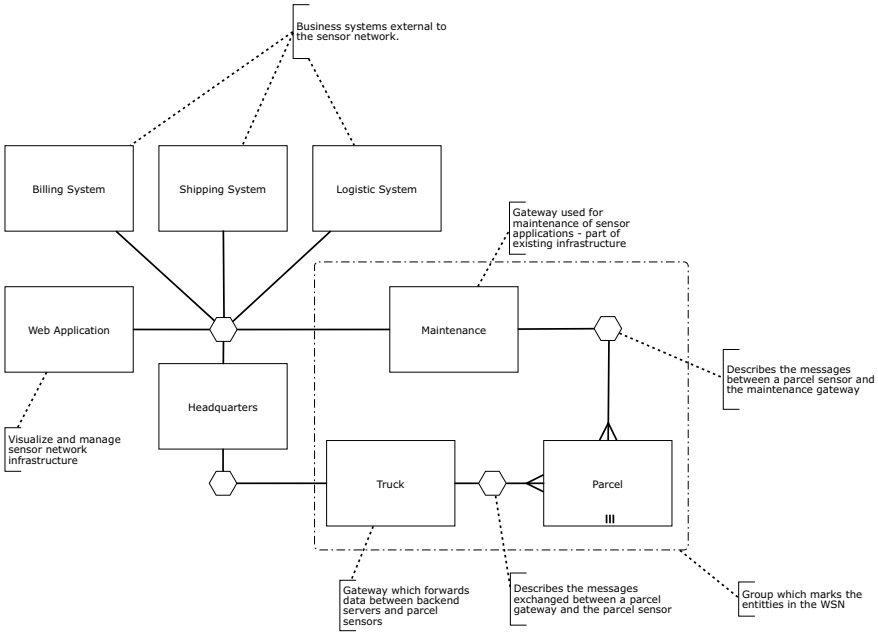


Figure 3.17: Complete system overview for the parcel delivery process.

delivery process, such as a Web-based interface for visualization and control, and business systems for billing, shipping, and logistics. To clearly separate the entities which are part of the sensor network these are grouped using a dotted-dashed rectangle, i.e., the BPMN group notation. Additional textual comments in the model itself provide descriptive information about the functionality of the entities.

Starting from the topology in Figure 3.16, we descend into refinements to show the process models at different levels of abstraction for the individual entities. Figure 3.18 is the first such refinement and shows the main overview of the parcel delivery process from the perspective of the company's headquarters.

3.3.1 The `Headquarters` Model

The `Headquarters` process in Figure 3.18 is an example of a classical BPMN model corresponding to the Level 1 and Level 2 abstractions (cf. Section 1.4). Also note that some of the tasks involve manual execution. The diagram shows the three pools `Headquarters`, `Truck` and `Parcel`. The `Truck` and `Parcel` pools are collapsed (i.e., process details are not visible) for visual compactness.

The `Headquarters` pool is expanded and shows the process controlling this pool which first checks the quality of the set of goods that are to be delivered. The task `quality control goods` therefore updates the data item called `quality`. The direction of the association flow indicates that the task writes to this data item changing its state. The same data item is addressed by the condition of the exclusive gateway by name, thus the goods are exchanged if their quality is not good.

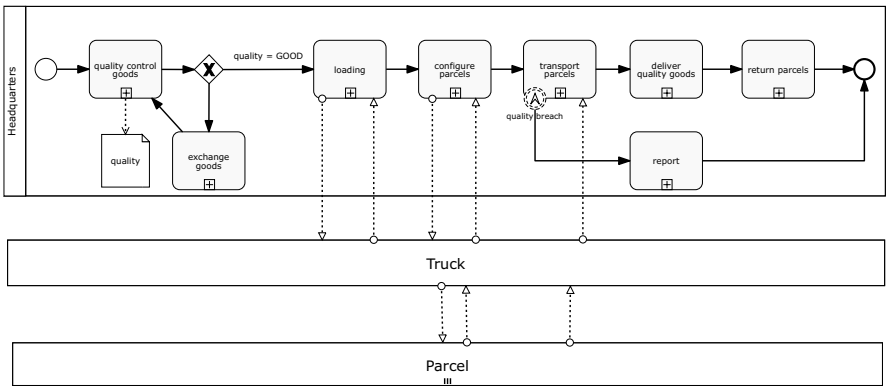


Figure 3.18: Classical BPMN model defining the topmost process which is manually performed at the `Headquarters`.

Otherwise, the goods are put into parcels and the parcels are loaded on the truck. As denoted by the communication flow, during the `loading` sub-task, messages are exchanged between the headquarters and the truck. The details of this communication flow are defined by the refinement models of the engaged tasks. On the big picture, the headquarters inform the truck about the loaded parcels and their destination and the truck acknowledges this information. Next, the parcels are configured with the temperature thresholds they should monitor via communication relayed by the truck.

During the transportation of the parcels, a `quality breach` event can occur which is denoted by the two concentric circles on the border of the task. This event is caught on the border of the `transport parcels` task because the respective symbol is hollow. Also note that the dotted lines indicate that the occurrence of the event does not terminate the respective task. Instead, the attached sequence flow is executed in parallel whereas the quality breach is only reported and the sequence flow ends as denoted by the simple end event. As soon as the truck arrives at the destination, all goods for which no quality breach occurred are delivered to the customer. Last, all parcels which could not be delivered are returned to the headquarters which completes the whole business process.

3.3.2 The `Parcel` Model

The `Parcel` process model in Figure 3.19 is executed on the WSN nodes which monitor the goods in the parcels. This model is the refinement of the collapsed pool with the same name from the previous model (cf. Figure 3.18). This process model represents a key contribution of this thesis: business analysts describe reactions to business events on an abstract level and hereby control the actual execution of the business process on the sensor nodes.

The process controlling a parcel with a WSN node first performs some initialization and executes the configuration task next. The `threshold` data store is associated with the `configuration` which persists the respective value. Thus, the WSN node, upon a system reboot, can restore the configuration from the

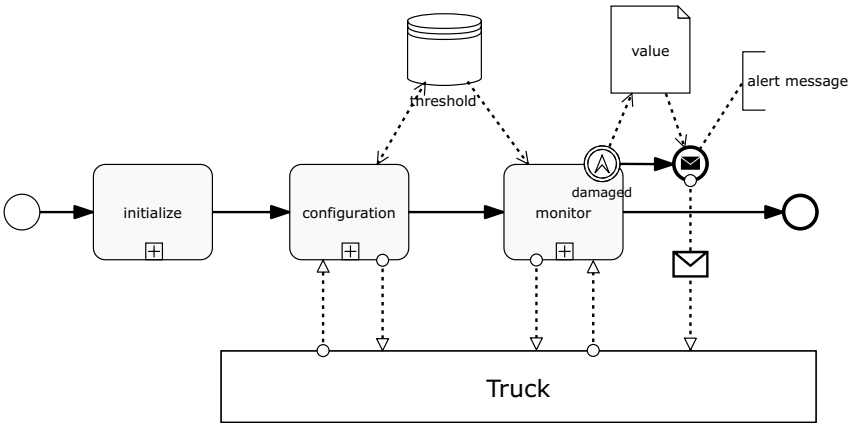


Figure 3.19: Model of the `Parcel` which is created by business analysts. Code generated from this model is executed on a WSN node.

data store instead of spending energy to communicate with the `Truck`.

After configuration, the `monitor` task is executed. This task can read the `threshold` value and has also a catching event attached on its border. In contrast to Figure 3.18, the solid lines of the concentric circles indicate that the occurrence of the `damaged` event does terminate the `monitor` task. The fact that the envelope of the subsequent end event is filled indicates that this event is a throw event, the counterpart of a catch event. Thus, the transmit (or throw) message event is the source of a communication flow.

The transmitted message contains the sensor value from the `damaged` event and the attached comment informs that this message is an "alert message" which signals to the truck that a quality breach occurred. The envelope on top of the communication flow arrow designates that, in contrast to previously mentioned communication flows, this flow represents an actual communication definition. The properties of the envelope component define the details such as the communication protocol and the payload format that should be used for this message.

3.3.3 The `monitor` Model

The `monitor` task from the `Parcel` model is further refined by the model shown in Figure 3.20. This model is an example which is created by software developers because it deals with technical details such as timers and heartbeat messages.

The `monitor` task reveals some additional details for the model which describes the behavior of the WSN node. First, the sequence flow coming from the start event immediately flows into an intermediate timer event, `in 5 s`, which defines the sampling rate. Here, the process semantically waits for the specified time span to elapse, afterwards the outgoing sequence flow is activated.

The following task samples the current temperature and, if the value exceeds the `threshold`, a `damaged` event containing the sampled sensor value is thrown. Otherwise, the `heartbeat` task signals to the truck that the system on the

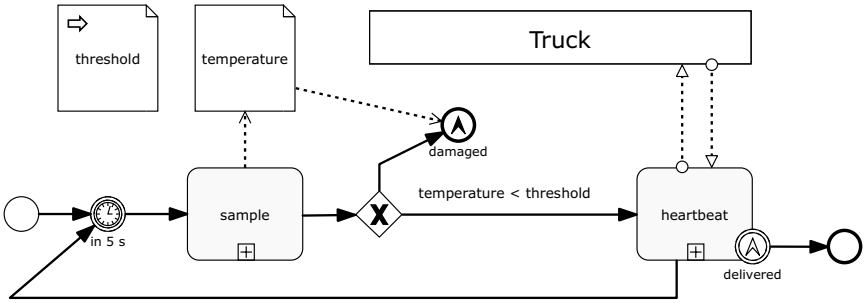


Figure 3.20: Refinement model of the `monitor` task which is created by software developers. Code generated from this model is executed on a WSN node.

sensor node is still functional. When the truck has arrived at the destination it responds to the heartbeat message with a shutdown notification which is propagated by means of a `delivered` event. If this event occurs, the sequence flow reaches an end event which in this case completes the `monitor` process. If the messages exchanged during the heartbeat task do not signal a completed delivery the `hearbeat` task completes normally and control follows the outgoing sequence flow. In this case, the `monitor` process loops back to the timer event and subsequently samples the new temperature value.

Summary

In this chapter, we introduced a BPMN modeling style for describing WSN applications with sufficient execution details. We showed how the heterogeneous, reactive, and communication aspects of WSN can be captured using BPMN models. This allows WSN to be incorporated into business processes models from the design phase. In the next chapter, we will focus on the transformation from model to executable source code for WSN.

Chapter 4

From BPMN Models To WSN Executables

Specifying detailed models is the first step towards a business-aligned implementation of processes. In this chapter, we describe our compilation algorithm which automatically transforms BPMN models into executable applications for WSN. This transformation represents one of the major contributions of our model-driven methodology. The input models are assumed to adhere to the style and rules described in the preceding chapters.

We first describe the general algorithm, which assumes that models contain only singleton sub-processes, i.e., there is at most one instance for every enclosed sub-process. We refer to such models as being *static* as they can be compiled to a list of static methods and variables which capture the execution of the process. Consequently, this code generation mode produces very efficient code with a low execution overhead.

In practice, models may contain multiple instances of the same sub-process executing simultaneously. We refer to such models as being *dynamic* because they require a run-time environment which manages the state of the individual sub-processes. We present our micro run-time which ensures the proper termination of sub-processes as well as the propagation of events according to the BPMN execution semantics.

To benefit from the efficiency of the code generation in static mode, we further describe an algorithm that is able to detect whether a model is always static. Moreover, we discuss how the two code generation modes (static and dynamic) can be combined.

4.1 Execution Semantics

Before we describe the code generation algorithm, we revisit the BPMN execution semantics. From an abstract perspective the control flow of a BPMN process can be viewed as a directed graph. The nodes in the graph are represented by the BPMN components, namely events, sub-processes, activities, and gateways. Whereas the BPMN sequence flows represent the graph's directed edges. Start events represent sources, whereas end events are sinks. Moreover,

the graph may contain cycles as well as disconnected components, namely event sub-processes.

4.1.1 Activation and Logic

Akin to Petri nets, BPMN uses tokens to conceptually describe the execution of processes. Tokens travel along the edges of the graph from sources to sinks following the direction of edges. When one or more tokens reach a node, the node is *activated* and executes some *logic* which is dependent on the type of node. After execution, tokens are placed on outgoing edges. A node's logic defines how many tokens from the incoming edges it requires for activation and whether tokens are placed on each of the outgoing edges or only on a subset thereof. Figure 4.1 depicts three different executions states of a model using tokens.

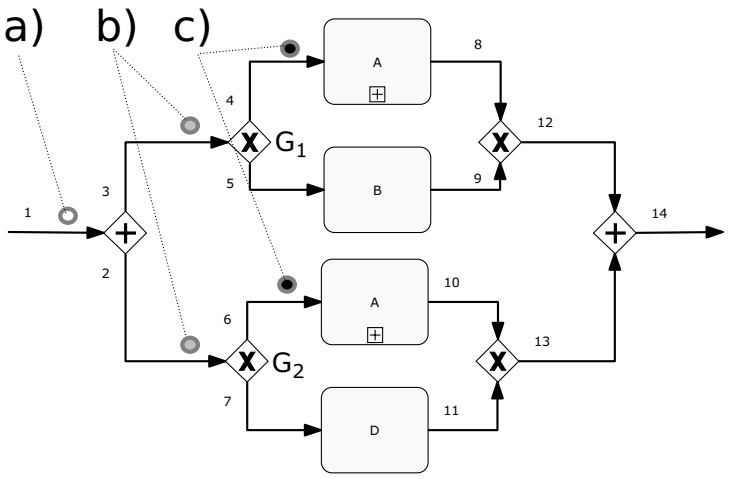


Figure 4.1: Example showing parallel instances for the same sub-process `A`. The current state of execution which is marked by conceptual tokens: a) in the first execution state a token enters the process, b) in the second execution state, the token is split at the parallel gateway, c) and in the third execution state, the tokens travel according to the conditions at the XOR gateways.

When a process is activated, the respective token is placed at the empty start event. The execution logic of a start event is void and the token is simply placed on the outgoing edge. Furthermore, whenever a token reaches a process or a sub-process a new instance thereof is created. The sub-process then executes its internal specification and upon normal completion, the token is placed on the sub-process's outgoing edge. A sub-process or task completes normally if no enclosed activities are still executing and there are no triggered events. A sub-process or task may also complete exceptionally upon the occurrence of an interrupting event. In this case, a token is placed on the exception flow outgoing from the corresponding catch event on the border of the sub-process or task.

The logic of a parallel split gateway, with a unique incoming edge and multiple outgoing edges, only requires one token on the incoming edge and will

generate one token for each of the outgoing edges. Conversely, the logic of a parallel merge gateway, with multiple incoming edges and a single outgoing edge, will wait for at least one token on each of the incoming edges. Only when all edges delivered a token the node fires one token on its outgoing edge. For catch events the execution logic specifies to wait until the occurrence of the event, which is then consumed. Subsequently, a token is placed on the outgoing edge. Conversely, throw events generate the respective event and continue by placing a token on the outgoing edge.

4.1.2 Parallel Instances

Two instances of the same sub-process can potentially be activated simultaneously at the same or at different locations in the process graph. We refer to such a case as *parallel instances*. Figure 4.1 shows such an example with two parallel instances of the sub-process `A` at different locations in the graph. This case is reached by executing the following paths $\{1, 3, 4\}$ and $\{1, 2, 6\}$, according to decisions at the exclusive gateways $\mathtt{G}_1$ and $\mathtt{G}_2$.

To support a BPMN-compliant execution, the sub-process `A` must be re-entrant. A *re-entrant* sub-process can be safely invoked again before its current execution is complete. This property also ensures that the state of a process is completely independent from other parallel instance of the same type.

Another situation where re-entrant processes are required is recursion. This concept is important because it allows expressing solutions for certain problem types in a compact manner by referring to previous states. The BPMN standard per-se does not prohibit recursive models. The specification states that when a token reaches a sub-process a new instance of this process is instantiated which is independent from the other instances of the same sub-process. However, in practice such models lack synchronization according to the definition from [41]. Nevertheless, we choose to support such models as they might occur in the specification of WSN protocols or different utility methods such as filters.

Figure 4.2 shows an example for a recursive sub-process `A` which first increments a counter specified by the `count` data item defined in the enclosing process. Based on the value of the counter the process then either throws an escalation event `E`, an escalation event `F`, or simply ends normally. After only a few iterations, the event `E` is thrown, and there are three active instances of sub-process `A`. The question now is which instance of `A` will be terminated when the interrupting event `F` occurs? The BPMN standard is silent in this respect. There are two possible options: terminate all instances, or terminate only the last instance. In both cases, the upper conditional path, corresponding to the `count` value being greater than three, is never executed. We believe the semantic which terminates all instances is more intuitive. However, from a technical perspective, it is possible to implement both approaches.

Static Versus Dynamic Models

In a *static* model all enclosed sub-processes are singletons. In this case, it is not required to separate the state of sub-processes, and it is not necessary for sub-processes to be re-entrant. As a consequence, the entire model structure can be flattened into a single name space with static methods and variables. Further optimizations are possible in this case with respect to passing of results

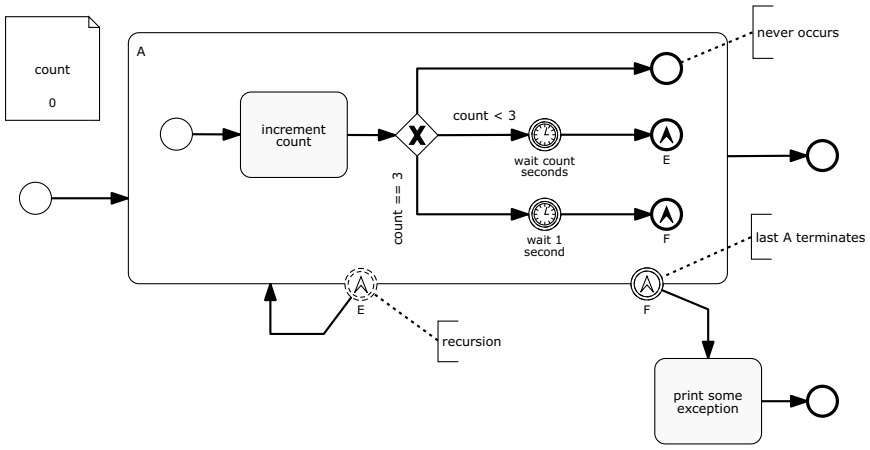


Figure 4.2: Example of a recursive sub-process. The non-interrupting event `E` causes multiple instances of sub-process `A` to be started. The `count` data item avaialable in the enclosing (parent) process acts as a global variable and is used in the condition which specifies the case which ends the recursion.

and accessing variables in the parent process. By contrast, a *dynamic* model contains potentially multiple instances of the same sub-process executing in parallel at the same or at different locations in the process. In this case, the BPMN execution semantics requires the different instances to have their own state.

Supporting parallel instances and re-entrant sub-processes has certain implications for the code generation. We can distinguish between the following two cases. The first case is when the maximum number[1] of parallel instances for a given process is known at compile time. In this case, it is possible to use a modified static code generation which duplicates methods and variables according to the maximum number of instances. This provides an independent execution state for a fixed number of instances at the cost of memory space. However, such a solution is inefficient for a large number of parallel instances. The second case is when the maximum number of parallel instances for a given sub-process is not known at compile time. The number of parallel instances is thus dynamic and depends on the execution state of the process.

To enable a dynamic solution, a run-time environment is required which manages the instantiation of an arbitrary number of sub-processes. The run-time must allocate the required stack and heap memory for the state of each sub-process instance. In this case, methods and variables are no longer static but associated with the particular instance of a sub-process. Moreover, the run-time is responsible for correctly propagating events, continuing control flow when a sub-process instance completes, as well as terminating all sub-processes when the enclosing process is interrupted.

Consequently, the compilation algorithm must be able to deal with both

[1]The upper limit for number of instances is also bound by the memory resources of the underlying platform.

the dynamic and the static case. We start by presenting the general algorithm for the static compilation mode. The dynamic compilation mode is based on the same principles as the static case. In addition, the dynamic mode adjusts the generated code to properly interact with the run-time. The adjustments address starting new process instances, passing parameters and return values, and accessing data. In general, because no run-time is involved, the static case is more efficient and exhibits a lower execution overhead.

4.2 Compiler Algorithm

In this section, we describe our code generation algorithm which transforms a consistent (and partially sound) BPMN model into executable code for a given WSN platform. Before delving into the details of our approach, we first fix the consistency and soundness pre-requisites. Following, to position our contribution, we review the state-of-the-art methods for compiling business process models to different executables representations.

Soundness

There are several, albeit similar definitions of model *soundness* in the literature such as the one in [141]. A sound model does not have a "lack of synchronization" and is "deadlock-free" as defined in [41]. These conditions state that it is not possible for two sequence flows to execute the same part of the model at the same time, and that there is always at least one sequence flow executing.

Using the token analogy, the synchronization rule states that, at any point in time, there is only one token traveling along a sequence flow. Whereas the deadlock-free rule states that there is at least one token left in the model. Intuitively models which do not exhibit a lack of synchronization are equivalent to static models as defined previously. The soundness criteria ensures the models are correct in terms of execution semantic.

Consistency Rules

However, the soundness criteria described above are too general and models which pass them still lack the information required for an automatic transformation to executable code. Thus we introduce several *consistency rules*, which ensure the available information is complete and consistent. While we consider the deadlock-free property to be a necessary requirement, we do not consider the lack of synchronization property because we want to support dynamic models (cf. Section 4.1.2). Instead, in addition to the deadlock-free, we require models to pass the following consistency rules. All rules can be statically checked before compilation.

- We require sub-process to have a unique entry point. In other words, multiple empty start events are not allowed. Also, multiple non-empty start events must be synchronized using event-gateways (cf. Section 3.1.3).
- In general events, activities, and messages require labels. The label of timer events must respect the time predicates specified in Equation 3.1. For matching throw and catch events, the label must be present for escalation and signal events.

- All basic activities must specify a string reference to an existing platform API (cf. Section 3.2.2) operation. Furthermore, all sub-processes must contain a valid reference to an existing model.
- The data inputs and outputs which are required for execution must be fully specified for respective activities and sub-processes.
- All strings used for referring to model elements must have a valid correspondence. In addition, conditions on sequence flows and code snippets in script tasks code must be specified using a valid Java or C# syntax.
- If a model uses communication flows, the addressing must be specified according to the rules described in Section 3.2.3. Furthermore, if messages are present they must be attached to communication flows and must specify a payload structure.

4.2.1 State of the Art

As described in Section 2.1, graphical models for business processes are specified in languages such as BPMN or the UML activity diagrams and use the concept of directed graphs. Such visual languages are often called *graph-oriented* as they use "unstructured continuations in the control flow" [65]. By contrast, BPEL models and programming languages such as C are *block-oriented* using structured blocks for control flow.

Compilation techniques from visual, graph-oriented languages such as BPMN to executable, block-oriented languages such as BPEL have been extensively studied [65, 66, 175, 186]. The main goal of such transformations is to close the gap between models and executable processes for a model-driven development in a SOA context. There is even work which deals with the opposite transformation [107], from block-oriented to graph-oriented languages, which may be used for example to document processes which are already implemented. Some of the transformations are partial [180] or require manual adaptations [134], some focus on producing readable BPEL code [168], while others [133, 134] require specific BPEL language constructs such as event-handlers and links. A recent report [65] exhaustively describes a transformation from graph-oriented to block-oriented models. The approach complements the original method proposed in [66, 186] based on compiler theory and "goto elimination". The transformation focuses on arbitrary, unstructured cycles which occur in graph-oriented languages and how these are mapped to structured constructs used by block-oriented languages.

The existing transformations are sophisticated but focus mainly on the problems arising with the control flow. Data, communication, and events, which are required for implementing executable processes which capture the distributed and reactive aspects of WSN applications, are not discussed. Also, multiple instances of the same sub-process are not treated. Hence dynamic models, as defined previously, are not supported. In addition, efficiency, in terms of resource usage, is not the main concern of the existing transformations. The process specifications are intended to execute on servers with sufficient memory, sufficient computational power, and a constant power supply.

For our WSN application domain such general approaches cannot be used directly. Either an efficient BPEL engine or an additional BPEL to code transformation are required. The first option is not viable as discussed in Section 2.2.

The second option is as challenging as the direct BPMN to code transformation but with limited control over the intermediate BPEL code. We therefore choose a direct BPMN to code transformation to meet the limited resources of our target platform.

4.2.2 Our Contribution

The novelty of our compilation approach is threefold. First, we do not use an intermediate representation such as BPEL. Instead, models are compiled directly to executable code expressed in a structured, typed programming language such as C, Java, or C#. Second, we reduce the original process graph to a set of sub-graphs which are connected by abstract goto jumps. This reduction is possible because our programs execute in an event-based run-time. Intuitively, the abstract goto jumps are event handlers which ensure execution continues at the correct point. Because of this reduction, the sub-graphs tend to contain a small number of components. Consequently, the chance for cycles to occur in a sub-graph is reduced. Last, we deal with the communication and data aspects, event propagation, as well as possible multiple instances of the same sub-process.

For the sub-graphs which do not contain cycles, our transformation to block-oriented programs uses code patterns similar to the ones described in [134]. In addition, our generated code must implement the execution semantics which a process engine would implement, e.g., for the various synchronization mechanisms and event propagation. For the remaining sub-graphs, which do contain cycles, we reuse the PST technique [171] for extracting the structure of a sub-graph. Afterward, the sub-graph can be transformed as described by [66] to obtain executable code snippets which are inserted at corresponding locations in the main code block.

Algorithm 1 shows an overview of our transformation. At this stage no assumptions are made about the model, which can be both static or dynamic (cf. Section 4.1.2). In the first step, the compiler creates a symbol table which contains the signatures of all sub-processes and all events used in the entire process hierarchy, starting from the top-most level. In the second step, the compiler parses each sub-process individually and creates for each a set of methods and a set of variables. The set of methods for each sub-process contains at least the `start`, `stop`, and `pending` methods. The `start` method represent the entry point of a sub-process which corresponds to the sub-graph beginning at the unique, empty start event. Each method is represented as an abstract syntax tree (AST) which contains corresponding statements. The statements in the AST are populated using a recursive `EXTRACT-METHOD` algorithm.

critical nodes

The core of the compilation is based the fact that asynchronous operations and synchronization gateways represent critical points where execution can be decoupled. These decoupling points we refer to as *critical nodes* and define the entry and exit nodes of sub-graphs. The following components represent critical nodes in a process graph: intermediate events such as timers, send and receive events or tasks, merge gateways, and asynchronous tasks. A sub-process is itself critical if it contains critical nodes. Thus for each critical node, additional methods are created and added to the set of methods for each sub-process.

In a first approximation, partitioning a process graph into sub-graphs (corresponding to methods) can be achieved by a breadth first search traversal starting from each critical node. If during the traversal a critical node is reached, the

Algorithm 1 Overview of the main compilation algorithm which decomposes the hierarchy of models starting at the top-most process model. The model for each sub-process is parsed to create a set of variables, and a set of methods. Methods contain statements which are organized in an AST.

Require: Top-most level process model T
Ensure: Map V containing the set of variables for each sub-process model
Ensure: Map M containing the set of methods for each sub-process model
  $P \leftarrow$ all models referenced in the hierarchy starting at T
  {Phase 1}
  for each p in P **do**
    $N[p]$ {set of critical nodes} $\leftarrow$ all critical nodes in p
    $s \leftarrow$ signature of p (input, output, throw events)
    $S[p]$ {signature table} $\leftarrow S[p] \cup s$
  end for
  {Phase 2}
  for each p in P **do**
    $V[p]$ {set of variables for p} $\leftarrow$ all data items in p
    i {entry node} $\leftarrow$ find unique start event of p
    $m \leftarrow$ EXTRACT-METHOD(i) {the `start` method}
    $M[p]$ {set of methods for p} $\leftarrow M[p] \cup m$
    for each n in $N[p]$ **do**
      if n invokes a sub-process s **then**
        Check that the invocation signature matches $S[s]$
        Resolve catch events in p to events thrown inside s
      end if
      $m \leftarrow$ EXTRACT-METHOD(n) {for the sub-graph starting at n}
      $V[p] \leftarrow V[p] \cup$ SPECIAL-VARIABLES(n)
      for each incoming edge e into n **do**
        for each method μ containing e **do**
          Attach SPECIAL-STATEMENTS(n) to the AST of μ
        end for
      end for
      $M[p] \leftarrow M[p] \cup m$
    end for
  end for
  for all inclusive OR merge gateways and multiple end events **do**
    Adjust semaphores in the AST of methods corresponding to split gateways
  end for

traversal backtracks. Before backtracking, the `START-STATEMENTS` for the critical node must be added to the AST of the current method. These statements depend on the type of node and are generated using macros. For example in the case of asynchronous operations, the start statements must first register a callback handler with the OS, then trigger the corresponding operation. In this case, the start statements also deal with the dynamic and static cases. In contrast, basic (non critical) nodes simply add their `EXECUTION-STATEMENTS` to the AST. These can be simple arithmetic operations or synchronous API calls. The previously populated symbol table is used to check that an invocation to a sub-process matches the respective signature. Furthermore, the table is used to

match events thrown in a sub-process with the corresponding catch events on the higher process levels.

After extracting the AST, customized macros create `SPECIAL-VARIABLES` and `SPECIAL-STATEMENTS` which again depend on the node type. The macros are platform specific and deal with the intricate details of configuring timers, packing and unpacking communication messages, as well as setting and clearing flags used for synchronization. These statements do not always belong to the currently extracted AST, and may require adjusting the AST of different methods. For example, when receiving a message, the received data must be unpacked before processing. The statements which unpack the data must be inserted at the top of the AST of the reception callback handler.

Because BPMN models may contain cycles, extracting the AST from a cyclic sub-graph is a challenging task. Detecting and correctly extracting the structure of loops is necessary before a transformation is even possible. Fortunately, several techniques allow to extract structured loops, such as the familiar `while` and `for` constructs in imperative programming languages, from arbitrary process models. For example, the refined process structure tree (PST) [171] returns "single-entry, single-exit regions" which can be used for pattern-based automatic code generation such as the one presented in [65].

Algorithm 2 EXTRACT-METHOD(n) creates an AST corresponding to each method which starts at a critical node.

Require: Global *visited* map, initialized to $visited[i] = false$, $\forall i$ nodes
  Initialization happens $\forall$ invocations from Algorithm 1
Require: n current traversal node
Ensure: S containing statements arranged in an AST
  if $visited[n] = true \wedge n$ is not critical **then**
    $S \leftarrow$ structured/unstructured loops from the PST algorithm on the sub-graph of all visited nodes ignoring outgoing edges from critical nodes
    return S
  end if
  visited[n] = true
  if n is critical **then**
    $S \leftarrow S \cup$ START-STATEMENTS(n)
    return S
  else
    $S \leftarrow S \cup$ EXECUTION-STATEMENTS(n)
    for each (n, m) outgoing edge from n **do**
      $S \leftarrow S \cup$ EXTRACT-METHOD(m)
    end for
  end if
  return S

Hence, for parsing arbitrary models, which contain cycles even after splitting the initial process into sub-graphs, we employ a hybrid approach as shown in Algorithm 2. In this case, nodes are marked as visited during the traversal. The list of visited nodes is re-initialized each time `EXTRACT-METHOD(n)` is invoked from Algorithm 1. Hence, the list of visited nodes is only valid for one recursive traversal. In case a visited node is encountered, which is not a critical node, we use the PST algorithm to extract the corresponding structured loops. The

input for the PST algorithm is the list of currently visited nodes, ignoring any edges outgoing from critical nodes.

4.3 Compilation Patterns

Both the compilation and parsing algorithms uses several macros to generate different code patterns based on the type of node. Table 4.1 gives a brief textual description of these macros. For each individual node type, the generated code patterns are described in detail in the following sections.

- EXECUTION-STATEMENTS(n)
  Generates method calls. For a task, the signature is checked against the platform API. The data inputs and outputs are matched, the data associations (if any) are resolved, and the corresponding operation is invoked with the specified input parameters according to name bindings. For throw events, the catch event is resolved using the process hierarchy which translates into a corresponding method call.
- START-STATEMENTS(n)
  Generates the statements required to start a critical operation. For an asynchronous operation, the continuation pointers are created. Subsequently, the operation is started by calling the respective method. For a gateway, the merging function is invoked with the correct semaphore identifiers.
- SPECIAL-VARIABLES(n)
  Generates declaration code required for merging variables at gateways, timer variables for corresponding events, as well as buffer space for communication and persistent data.
- SPECIAL-STATEMENTS(n)
  Generates the statements required by special access functions to encode communication messages as well as packing and unpacking persistent data. Moreover, these statements are required for managing temporary helper variables, various activation flags for timers and communication, as well as clearing and setting of states in synchronization semaphores.

Table 4.1: Overview of macros which generate code patterns for a given node n based on its type.

To keep the code patterns generic, we use C as an example programming language due to its widespread usage and familiar syntax. However, our compiler generates code for the Mote Runner platform where programs are written in Java and C#. Nevertheless, the back-end of our compiler can be adapted to generate code for other widely used WSN platforms such as TinyOS (nesC, C dialect) or Contiki (C++). In the code example, we emphasize the functions required from the platform API using underlined fonts.

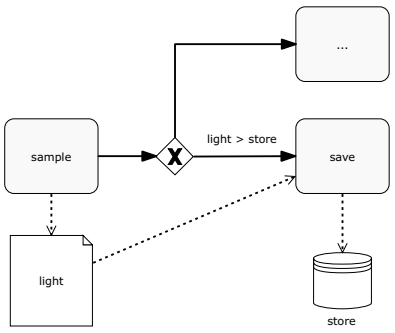
4.3.1 Data Flow

Data items in the process model become variables in the generated source code. How the variables are accessed (static, objects) and what visibility (private, public) they have depends on whether the static or the dynamic compilation mode is used. For example, in the static case, all variables are defined static and are publicly visible.

The data associations to tasks and sub-processes are converted to read and write operations to variables. By contrast, data stores require some additional processing. First, they need to be placed into persistent memory. In our transformation, the `PERSISTENT` space is allocated as a `byte` array which accommodates the size of the data store. As a consequence, such arrays cannot be accessed directly and require some temporary variables. Second, the access operations for data stores are particular to a certain platform API which is able to read and write (`persistent_pack` and `persistent_unpack`) from persistent memory such as flash or EEPROM.

The type of variables can be specified using the BPMN `category` property. The type of data objects can be both basic (e.g., `int` and `long`) or complex data types. For data objects which do not specify a type, it can be inferred either by analyzing the return value of operations or the name binding of input and output parameters. The BPMN `Assignments` property as well as the data flows are used in the type resolution process. Data objects whose type is not defined and cannot be inferred will trigger a compilation error at the model level. A model compilation error also occurs if a data item explicitly defines a type which conflicts with the inferred one.

An annotation with parallel vertical lines (||||) on a data object signifies that there are multiple instances of that object. We treat this case as an array with elements of the specified type. The individual elements can be accessed using the normal programming syntax. For example $d[2]$ will access the second element in a multiple instance data object d.




```
1 PERSISTENT byte store[2];
  int light;
3
  void process (){
5   light = sensor_sample(LIGHT);
    int tmp = persistent_unpack(store,0);
7   if (tmp > THRESHOLD) {
      persistent_pack(store,0,light);
9   }
    else {
11    ...
    }
13 }
```

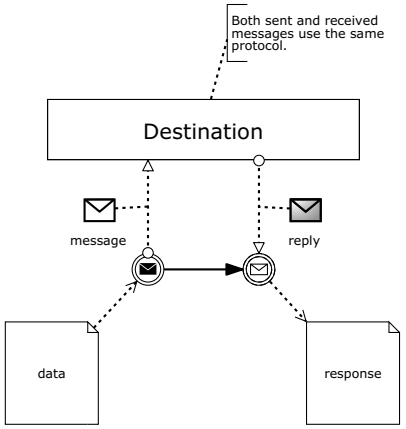
Figure 4.3: Example code pattern generated for persistent and volatile data.

The left side of Figure 4.3 shows an example process which performs various operations on data objects. The right side shows the example code pattern which is generated from the model. The example first samples a `light` value which is then compared (line 7) with the persistent `store` value. For this purpose an intermediate variable `tmp` in volatile memory (line 6) is required. If the last

sample is greater than the stored one, the new value is saved persistently (line 8) using the respective platform API.

4.3.2 Communication Flow

Similar to persistent data stores, generating code from the communication flow specified in process models requires several steps. First, enough memory must be allocated to accommodate the message header and payload. In our case, byte arrays are used to represent the raw serialized messages. Second, the compiler ensures that the task or event which sends a message matches the protocol specification for the communication flow and the corresponding tasks on the receiver side, a protocol mismatch represents a compilation error. Additionally, the header of the message is populated based on the address resolution mechanisms described in Section 3.2.3 using helper methods (`protocol_header`) from the platform API. Moreover, for sending and receiving messages, each task which sends, respectively receives messages must provide a `protocol_pack`, respectively `protocol_unpack` method. These methods are part of the platform API and they encode respectively decode the message payload fields which are specified in the model as part of the BPMN message.




```
1  byte message[10];
   bool protocol_flag;
3  void (*continue_send)();
   void (*continue_receive) (byte *reply);
5  long data;
   int response;
7
   void process(){
9    protocol_header(message);
     protocol_pack(message,9,data);
11   continue_send = after_send;
     protocol_flag = true;
13   protocol_send(message, 10);
   }
15
   void after_send(){
17   protocol_flag = false;
     continue_receive = after_receive;
19   protocol_flag = true;
     protocol_receive();
21 }

23 void after_receive(byte *reply){
     protocol_stop();
25   protocol_unpack(reply,13,&response);
   }
```

Figure 4.4: Generated code patterns for message transmission and reception.

The left side of Figure 4.4 shows an example process model where a message event sends some `data` to a `Destination` pool. After the message is sent a reply, which contains some `response` data, is expected by the receive event. The right side of the figure shows the generated code pattern corresponding to the model. First the header for the message is populated (line 11) using the `protocol_header` method. Afterwards the `data` is encoded into the message payload (line 12) and the message is sent (line 15) using the respective protocol. For this particular protocol, sending is an asynchronous operation and execution

resumes at the `after_send` method as specified by the `continue_send` function pointer. Subsequently, the protocol receive mode is started (line 22) and when the reply arrives the `response` is decoded (line 27) from the message. While the protocol executes, it is marked as active using the `protocol_flag` to enable the correct synchronization at end events. Because the model specifies only a single reception is expected, the protocol is stopped (line 26) after the first message is received. In contrast to the `reply` buffer which is provided by the protocol receive callback (line 25), the `message` buffer is allocated statically (line 1).

4.3.3 Control Flow

Compared to data flow and communication flow, the translation of control flow is more complex.[2] To support standard sensor nodes, we have to address the fact that major WSN platforms are event-driven due to the efficiency benefits of this paradigm [22, 37, 68]. Thus, the compiler translates the sequential and parallel computation semantics of a BPMN model into equivalent event-based code.

Event-based platforms typically use a single *dispatcher* which constantly takes the next[3] *event* out of an event queue and calls the corresponding *event handler*. Event handlers may not block when waiting for something. Instead they only trigger a corresponding *asynchronous* operation and delegate the remaining processing to a *callback* which is a second handler that is registered at the run-time environment. By following this pattern the compiler maps each sequence flow to a chain of event handlers which are causally related to each other by the respective events. As long as an asynchronous operation has not yet completed, the dispatcher calls handlers which correspond to other sequence flows, thus processing multiple sequence flows virtually in parallel.

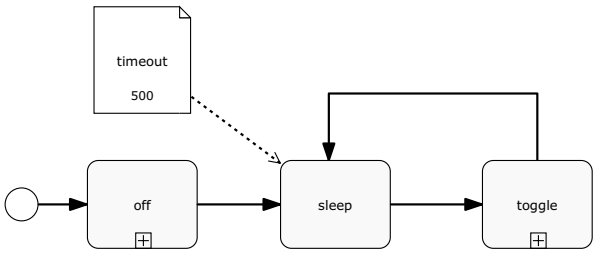
In our case, event handlers correspond to sequence flows outgoing from critical nodes (cf. Section 4.2.2) in the BPMN process graph. In other words, outgoing sequence flows of asynchronous sub-processes, tasks, intermediate events, non-empty start events, merge gateways, and event-based gateways represent all possible resume points for sequence flows at run-time. Thus, for each instance of these critical nodes, a function is generated which implements the corresponding logic according to the BPMN execution semantics (cf. Section 4.1).

Sub-Processes and Asynchronous Tasks

If a sequence flow reaches a sub-process, the generated code first calls the `start` method of the invoked sub-process. Furthermore, for every possible continuation, e.g., catch events on the border of the sub-process or outgoing sequence flows in the case of a critical sub-process, the pointers of the corresponding event handler functions are saved in extra *continuation variables*. These variables are managed in the same way as data item variables. Upon completion of the asynchronous sub-process, the generated code invokes the corresponding continuation function. Basic asynchronous operations, which are provided by the platform API and cannot be further refined, are treaded similarly. In this case,

[2] A succint transformation idea has been published in [21]. This section represents a comprehensive treatment of the different compilation patterns including synchronization primitives.

[3] The criteria which specify the next event can be defined in multiple ways. Often used examples are the oldest (FIFO) event, latest (LIFO) event, or based on priorities.



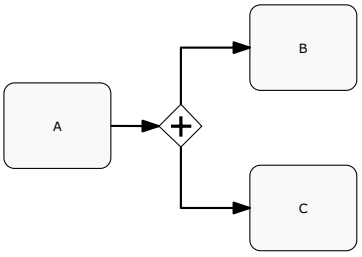

```
   void (*sleep_continue)();
 2 int timeout = 500;
   int toggle_status;
 4
   void blink_start() {
 6   off_start();
     sleep_continue = &after_sleep;
 8   sleep_timeout = timeout;
     sleep_start();
10 }

12 void after_sleep() {
     toggle_start();
14   sleep_continue = &after_sleep;
     sleep_timeout = timeout;
16   sleep_start();
   }
18
   void off_start() {
20   led_set_state(0);
   }
22
   void toggle_start() {
24   toggle_status = led_get_state();
     toggle_status = !toggle_status;
26   led_set_state(toggle_status);
   }
```

Figure 4.5: Model for a *Blink* application, and corresponding generated code, which toggles an LED every 500 ms. For simplicity, the models for the `off` and `toggle` sub-processes are omitted.

upon completion, the underlying platform run-time invokes the corresponding continuation function automatically.

The example code in Figure 4.5 shows the transformation of a corresponding model which uses several sub-processes and a basic task. The sub-processes `off` and `toggle` are non-critical because they only invoke the non-critical basic tasks `led_set_state` and `led_get_state` (line 20, 24 and 26; for simplicity not shown in the model). These tasks are defined as synchronous operations in the corresponding platform API. In contrast, `sleep` is a critical operation because the platform API provides an asynchronous interface. Thus the continuation pointer `after_sleep` is stored in the variable `sleep_continue` before the task is started (line 7 and 14). Furthermore, the `timeout` data item is associated with the `sleep` task, so that its value is passed to the operation (line 8 and 15). Because the `sleep` task is provided by the platform API, the `after_sleep` handler is automatically invoked upon completion.

Parallel Fork Gateways (AND-split)

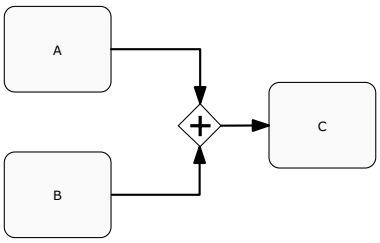



```
1 void process () {
    A_start();
3   // fork --- START
    B_start();
5   C_start();
    // fork --- END
7 }
```

Figure 4.6: Example code pattern for the parallel (AND) split gateway.

When reaching a parallel fork gateway, subsequent synchronous operations can be serialized, because WSN platforms typically only have one execution unit. To preserve a deterministic execution order we execute from top to bottom. In other words, the sequence flow which is geometrically closer to the top of the gateway rhombus in the process graph is executed first. If a critical node follows a sequence flow, the respective operation is started according to its asynchronous interface. Figure 4.6 shows a model and the corresponding code patterns which is generated in this case.

Parallel Join Gateways (AND-join)




```
1 #define MASK 0x03
  int semaphore;
3
  void process() {
5     semaphore = 0;
      A_start();
7     merge(0x1);
      B_start();
9     merge(0x2);
  }
11
  void merge(int token){
13    semaphore |= token;
      if (semaphore != MASK)
15       return;
      semaphore = 0;
17    C_start();
  }
```

Figure 4.7: Example code pattern for a parallel (AND) merge gateway.

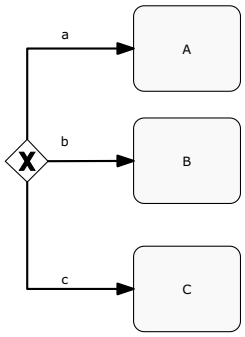
By contrast, the code which is generated for a merging gateway is more involved as shown in Figure 4.7. The generated code must check if all the sequence flows have been merged. Because a sequence flow may reach a gateway several times during the execution of the process we need to distinguish between sequence flows. Hence, the compiler uses a bit-wise `semaphore` for each parallel merge gateway and a corresponding `merge` function. To know whether a particular sequence flow has merged or not requires one bit of information.

Each sequence flow which ends at such a gateway invokes the `merge` method with a unique `token` with a single bit set. The tokens are assigned from top to bottom (geometrically) for all incoming sequence flows. The merge function continues only if all the bits in the semaphore `MASK` are set. The number of bits in the `mask` is given by the number incoming sequence flows. The semaphore bits are initialized to `0` when the process starts and after the gateway successfully merged the last incoming sequence flow. Afterwards, the following sequence flows are executed as described previously.

Exclusive Gateways (XOR-split and XOR-join)

When reaching an exclusive gateway which forks the control flow (XOR-split), only one of the outgoing sequence flows is activated based on the mutually exclusive conditions. An inconsistent model could have conditions which are not mutually exclusive. Also inconsistent are two or more unspecified (empty) conditions for outgoing sequence flows.

As in the case of parallel gateways, we again follow the top to bottom heuristic and execute only the outgoing sequence flow for the first condition which is satisfied. The condition must be specified either on the label of the outgoing sequence, which appears visually, or in the BPMN `condition` attribute, which appears in the properties editor. In both cases, the condition must be a valid Boolean expression, specified using a common programming language syntax, e.g., `threshold != 42`. To ease the specification of conditions, data items can be accessed by name. Additionally, the condition is parsed by the compiler and checked against the symbol table of variables and methods. In a sense, a condition is a snippet of executable code.




```
void process () {
  if (a) {
    A_start();
  }
  else if (b) {
    B_start();
  }
  else if (c) {
    C_start();
  }
}
```

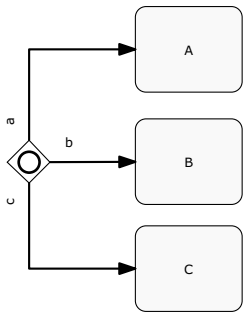
Figure 4.8: Example code pattern for an exclusive fork (XOR-split) gateway.

The left hand side of Figure 4.8 shows an example model which forks control flow using three conditions. The right hand shows the corresponding generated code using the `if-else` construct.

A special case is represented by exclusive gateways (XOR-join) which are used for merging, i.e., with multiple incoming sequence flows. These gateways are simply skipped and execution continues at the subsequent element. In this case, the code generation simply inspects the following sequence flow.

Inclusive Fork Gateways (OR-split)

A special kind of control flow gateways are represented by the OR gateways or inclusive gateways. OR-split gateways place a token on each outgoing sequence whose condition is met. In contrast to exclusive (XOR) gateways, the conditions of splitting OR gateways are not mutually exclusive. In general, OR-split gateways, can be replaced with multiple parallel fork (AND-split) gateways and exclusive (XOR-split) gateways which model all possible inclusive conditions. In practice, this is cumbersome and only leads to a larger model with a poor overview.




```
1  void process () {
     if (a) {
3      execute_A();
     }
5    if (b) {
       execute_B();
7    }
     if (c) {
9      execute_C();
     }
11 }
```

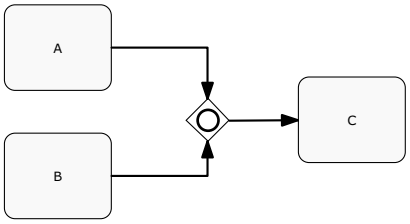
Figure 4.9: Example generated code pattern for an OR-split gateway.

Figure 4.9 shows an example code pattern which is generated for a split OR gateway. The difference to the code generated for the exclusive gateway in Figure 4.8 is that the conditions are not mutually exclusive and all branches could potentially execute. Thus, instead of `if-else` blocks, only `if` blocks are necessary, which allow multiple paths to be executed.

Inclusive Merge Gateways (OR-join)

By contrast, the semantic of an OR-join gateway (i.e., with multiple incoming sequence flows), is more involved. Informally, an OR-join gateway must wait for all incoming sequence flows included in an *upstream* path which can potentially carry an active token. Essentially, an upstream path follows the sequence flows backwards. Consequently, the execution logic of merging OR gateways cannot be decided using simple local conditions as in the case of parallel merge (AND) gateways. Different theoretical treatments of this problem as well as formal mathematical definitions of the execution semantics of OR gateways are presented in [35, 50]. The proposed solutions treat the problem exhaustively and assume a decision is taken at the merging OR gateway during run-time. Such a decision is costly as it involves an expensive state space exploration of all upstream paths.

However, if the process graph is compiled directly to executable code, as in our case, a state space exploration is only required once, namely at compile time. The state exploration at compile time yields all guard semaphores for which a gateway will wait for. The guard conditions are atomically modified at run-time by setting and clearing bits in semaphores at *key points* in the




```
1  int active = 0x00;
   int merged = 0x00;
3
   void process() {
5    active |= 0x01;
     A_start();
7    merge_gateway(0x1);
     active |= 0x02;
9    B_start();
     merge_gateway(0x2);
11 }

13 void merge_gateway(int path){
     merged |= path;
15   if (active ^ merged)
         return;
17    merged = 0x00;
      active = 0x00;
19    C_start();
   }
```

Figure 4.10: Example generated code pattern for an OR-join gateway.

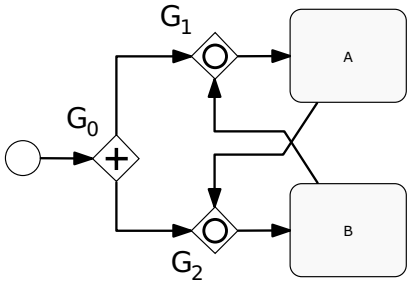
execution to indicate whether a path is active or has merged. These key points are represented by the other type of gateways which may create or absorb tokens, namely parallel gateways, event-based gateways, and OR gateways.

Consequently, we define an upstream path to stop at such a gateway, including the currently explored gateway, or a start event. Also, the path is not further explored in case a loop is encountered. Special dependency situations can occur if on the upstream path other OR gateways are present. The solution presented in [50] can be extended to output the key points where the execution must be recorded.

With these observations, our proposed merging code pattern is shown in Figure 4.10. The generated code uses semaphores which mark whether upstream paths are active or not. The upstream paths are assigned identifiers from top to bottom for each of the incoming sequence flows. Each OR gateway uses two semaphores where individual bits mark whether a path with the corresponding number is `active` or has `merged`. The semaphore bits are set and cleared at the key gateways during the execution. As in the case of parallel merge gateways, the state of the semaphores is initialized at the beginning of the process execution and after each successful merging.

In the generated code, the semaphores are prefixed with the name of the gateway. As in the case of AND-joins, the semaphores keep the information about the state of execution using a minimal encoding. The OR-join gateway will become active, i.e., will allow execution of the outgoing sequence flow, only if all paths which have been activated have also merged. Using Boolean operators, the above condition can be translated to an XOR operation with the two semaphores as operands. Only if the result yields a zero value the gateway will become active and execution will continue.

A typical example of interconnected OR gateways, which is cited often in the business process literature, is the one shown in Figure 4.11. This case may generate deadlocks as the two inclusive OR gateways seem to wait for each other. However, according to the semantics defined in [35] this does not represent a deadlock as the gateways are in “partial conflict”. Because we use the same




```
int G1_active = 0x00;
int G1_merged = 0x00;
int G2_active = 0x00;
int G2_merged = 0x00;

void process() {
  G0_split();
}

void G0_split() {
  G1_active |= 0x01;
  G2_active |= 0x01;
  G1_merge(0x01);
  G2_merge(0x01);
}

void G1_merge(int token){
  G1_merged |= token;
  if (G1_active ^ G1_merged)
    return;
  G1_merged  = 0x00;
  G1_active  = 0x00;
  G2_active |= 0x02;
  execute_A();
  G2_merge(0x02);
}

void G2_merge(int token){
  G2_merged |= token;
  if (G2_active ^ G_merged)
    return;
  G2_merged  = 0x00;
  G2_active  = 0x00;
  G1_active |= 0x02;
  execute_B();
  G1_merge(0x02);
}
```

Figure 4.11: Example code patterns for "dependent" OR-join gateways.

execution semantic, the example code pattern which our compiler generates also does not deadlock. Essentially, the process loops endlessly between the two OR gateways.

For the above example, assuming execution is performed from top to bottom, we have the following execution trace $\{G_0, G_1, A, G_2, G_2, B, G_1, A, \ldots\}$. The trace records whether the respective method in the generated code for a corresponding node is invoked, i.e., G_2 means `G2_merge` is invoked, and A means `execute_A` is invoked. Note that in the above trace, the method corresponding to the G_2 gateway is invoked twice. The first invocation does not activate G_2 because the gateway still waits for the lower sequence flow outgoing from G_0. The second invocation activates G_2 because both upstream paths which were active have now merged.

4.3.4 Events

Events are a core component of BPMN models. At the same time, they represent critical points where execution is decoupled, which is exploited by the code generation. For each BPMN event type (cf. Figure 3.4), the generated code must implement the corresponding semantic. Complexity is further increased as some events may *interrupt* either the sub-processes on whose border they are placed

or the enclosing process. Therefore each sub-process requires a `stop` method to abrubtly terminate all enclosed activities. For basic tasks, the platform API should ensure such methods exist.[4]

We first treat the code generation for the pair of throw and respective catch events. The empty (i.e., without a type specification) start and end events are treated separately. As a rule, non-empty (i.e., with a type specification) end events may only be throw events, whereas, non-empty start events may only be catch events. By contrast, intermediate events can be both catch and throw events. A special BPMN construct which reuses the code generation patterns for events is the event-based gateway, which we also treat separately.

delegates

As a general mechanism our compiler deals with events using function pointers or delegates. We assume the underlying run-time allows a pointer to a method to be registered for invocation. Function pointers represent an older concept, more recently introduced in object oriented languages by C#, where it is called a *delegate* [111]. "A delegate type describes the parameters and return values of the method and at run-time holds a pointer to an object and a pointer to a method implementation. A delegate concept allows an application programmer to tie event callbacks directly to those object instances that hold the application state."[22] For this purpose our compiler requires the delegate name, which defines the method signature, and the name of the method which implements the corresponding delegate.

basic vs. user events

We distinguish between *basic events* and *user events*. Basic events are mapped directly to a platform API, whereas, user events are defined in the process. Typically, basic events are timers, errors, and messages (using either default communication or a specific protocol). User events are escalation and signal events which are mapped to different delegates in a generic way.

For basic events the method name is automatically generated whereas the signature is given by the corresponding platform API delegate. The compiler must be able to uniquely identify such a delegate. For example, timer events will always map to the timer delegate specified by the platform API. As a concrete example in Mote Runner the signature for timer delegates is as follows:

```
public abstract void TimerEvent(byte param, long time)
```

As another example for basic events, consider message throw events which do not specify a particular protocol. In this case, the best available protocol on a corresponding platform is used. For example, in Mote Runner, the concrete signature for the transmission delegate is given by:

```
public abstract void TxDone(int flags, byte[] data, uint len,
  uint info, long time)
```

Message events which specify a particular protocol, will map to the communication delegates defined by the protocol. The throw events will map to sending delegates, whereas, the catch events will map to the reception delegates for the respective protocol.

By contrast user events allow to define custom delegates. Thus, the delegate name, which specifies the corresponding signature, must be present in the model properties. For escalation and signal events, the delegate name can be specified

[4] If such a method is not available, other platform specific alternatives to cancel asynchronous activities could be used. For example null pointers could be used to invalidate callback handlers.

in the `EscalationName` and `SignalName` properties as shown in Table 4.2. By contrast, BPMN error events map to Exceptions in corresponding `try/catch` blocks. In contrast to basic events, for improved mnemonics, we recommend that user events specify a label for the event which is visible in the graphical editor. If present, the label identifier is used for matching and as the method name during code generation. In case the label is absent and the event can still uniquely matched, a method name is automatically generated.

BPMN		Code
Event Type	Properties	Mapping
Signal	SignalName	Delegate
Escalation	EscalationName	Delegate
	EscalationCode	unused
Error	ErrorName	Exception
	ErrorCode	error code

Table 4.2: The correspondence between properties of user events and the generated code. Note that all BPMN properties are defined as strings.

For user events which specify no delegate, we employ a `GenericDelegate` as shown below. In this case, the event must have a label (visible in the graphical model) which allows to match throw and catch events. This consistency rule is enforced by the compiler.

```
  // generic delegate, which can be part of Platform API
2 namespace platform.api {
    // delegate signature
4   delegate void GenericDelegate(byte [] data);
  }
6
  // generated code for event implementation, label is used as method name
8 void func (byte [] input) {
    ...
10 }
```

The advantage of a generic delegate is that it can carry arbitrary parameters packed in a byte array. When a throw event is encountered the data items are packed into the array and when a catch event is reached the corresponding data items are unpacked from the array. This mechanism is similar to dealing with persistent data, as described in Section 4.3.1. Furthermore, if no results are passed the array is not used.

Empty Start Events

Start events which are empty represent the entry point to the `start` method of the enclosing sub-process. Start events which have an associated type, e.g., message, timer, require additional processing. They are treated as an empty start event followed by an intermediate catch event. Start events for event sub-processes are treated separately.

Catch events

In general, for each catch event c the compiler generates a corresponding method M_c. In the static compilation mode, the method M_C is also responsible for stopping the enclosing sub-processes if c has an interrupting semantic. The method

name is either automatically generated, for events without a label, or simply a canonical form (following the C naming rules) of the label for the corresponding BPMN element. The signature on the other hand is not graphically visible but is mapped according to the event type and its properties (cf. Table 4.2) of the respective event.

For basic events, the compiler must find the corresponding platform API operation which may trigger the event. If there is no such operation or there are multiple matches, the compiler generates an error. Otherwise, a unique platform API operation may potentially trigger the corresponding event. Thus, the compiler generates statements S_c for starting the corresponding operation and setting the continuation methods as described in Section 4.3.3.

User-defined catch events correspond to events which are thrown by enclosed sub-processes. Typically, these are escalation events. The generated code for these events simply sets a flag which marks that the corresponding event is active. The flag is cleared upon invocation of the method M_c.

Start Events for Sub-Processes

Start events for event sub-processes represent a special case of catch events. If a process P contains an event sub-process E the latter must begin with a single start event e, according to the BPMN specification (cf. Section 3.2.1). This event can only occur due to some action performed by a platform API function A. Consequently, A must be started for e to occur. The exact moment when A is started is defined by the following two cases.

In the first case, P contains a task with a reference to A. Consequently, E only becomes active as soon as a sequence flow reaches A. The compiler generates code which sets the continuation method to point to M_E before starting A. The second case is when P does not contain a reference to A. However, if there is only a unique match for A in the platform API, the compiler can generate code to automatically start A as soon as P starts as described above. The statements are inserted in the AST of the `start` method for P. Otherwise, for multiple matches, the compiler reports an error. In both cases, the compiler generates code for a method M_E corresponding to the sub-process E. The method contains the statements corresponding to the elements following the outgoing sequence flow from e.

Throw Events

For throw events the compiler simply generates code which calls the method (implementing the delegate) corresponding to the catch event. For user-defined events, the method can be resolved at compile time, in the static case, whereas a look-up may be required, in the dynamic case. In general, after the throw event, execution simply continues following the outgoing sequence flow.

An exception to the above rule are message throw events which are typically asynchronous operations (depending on the platform API). In this case, the throw event will first wait for the respective operation to complete, then continue execution. For message throw events the corresponding message catch event is always in a different pool. In a consistent model, the two events must be linked by a communication flow (cf. Section 4.3.2).

End events which specify a particular type, e.g., send, escalation, or signal, are treated as an intermediate, non-empty event followed by an empty end event. Consequently for such events, the corresponding catch method is invoked followed by an invocation to the process `pending` method. This way, we ensure all active paths have reached an empty end event.

Empty End Events

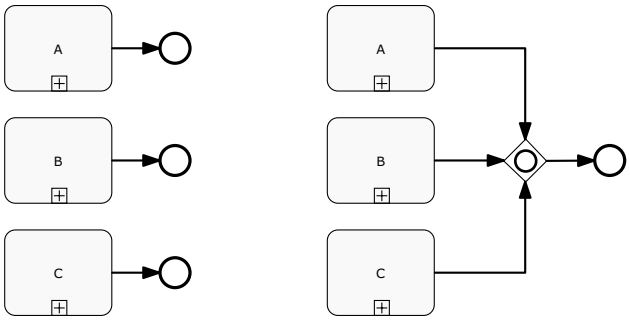


Figure 4.12: Multiple end events correspond to an implicit OR-join gateway. Thus, a process will complete only if all active paths have reached an end event.

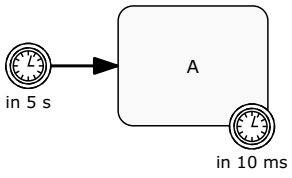
The BPMN execution semantics specifies that multiple empty end events are implicitly merged using an OR gateway as shown in Figure 4.12. Consequently, multiple end events can be treated using the code patterns described in Section 4.3.3. However, the flags and conditions required for merging OR gateways require considerable code space. In particular, the code space increases with both the number of gateways and end events.

A more efficient implementation for the process completion is to query whether any asynchronous operations are still pending upon reaching an end event. Thus, each sub-process has a `pending` method which is automatically generated by the compiler. The method inspects Boolean flags which mark whether asynchronous activities are still active. Some basic operation such as timers, depending on the platform API, may already have such flags. In this case, no additional code is required. Only if the `pending` method of a process returns true, the process finishes and execution continues at the place indicated by the continuation pointer.

Timer Events

A special type of catch events are timers which always catch the corresponding event and enable processes to specify real-time requirements which are vital for WSN application (cf. Section 3.2.4). Timer events are thus frequently used in processes. When placed on the border of an activity they are equivalent to a timeout.

During compilation, a timer event is translated to a variable (of type `timer_t` which is provided by the platform API) which keeps the state of the timer. Figure 4.13 shows the example code pattern which is generated for an intermediate




```
  timer_t timer;
2 bool timer_flag;
  const long timer_interval_5s = SECONDS(5);
4 const long timer_interval_10ms = MILLIS(10);
  void (*timer_continue)();
6
  process () {
8   timer_continue = &after_timer_5s;
    timer_flag = true;
10  timer_alarm(&timer, timer_interval_5s);
  }
12
  void after_timer_5s() {
14  timer_flag = false;
    A_start();
16  timer_continue = &after_timer_10ms;
    timer_flag = true;
18  timer_alarm(&timer, timer_interval_10ms);
  }
20
  void after_timer_10ms() {
22  timer_flag = false;
    A_stop();
24 }
```

Figure 4.13: Code generation example for an intermediate timer event used to trigger an activity at a specified time. The activity must finish within a certain timeout, as specified by the border event, otherwise the activity is abrubtly terminated.

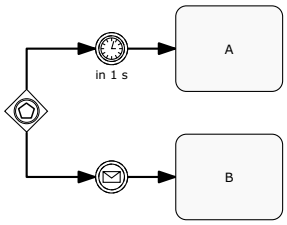
timer event which waits for 5 s, afterwards an activity A is executed. In addition, a flag marks whether the timer is active or not. Whenever the timer is started the corresponding flag is also set, and, when the timer fires, the flag is cleared. The flag is used by the corresponding `pending` method which checks whether a process still has some active operations. Depending on the platform API, the extra flag variable is not necessary if the platform API provides a way to query the activity state of the timer. Last, constants (lines 3 and 4) are required to hold the time spans specified according to the description in Section 3.2.4. The time intervals are transformed from portable units to constants using platform specific conversion (e.g., `SECONDS, MILLIS`) functions.

The same code pattern is used to generate code for start timer events as well as intermediate timer events which are placed on the border of activities. For events which are interrupting, when the timer expires, the `stop` method for the enclosing sub-process or attached sub-process is invoked (line 23) in addition.

Event-based Gateways

For event-based gateways, the code generation must first determine which activities may start the subsequent catch events. If there is a unique mapping, code is generated which starts the corresponding activities which will eventually trigger the respective events. The occurrence of an event selects the branch which will be taken. The other branches are never executed. Consequently, the generated code cancels all state related to the activation of the complementary events as soon as one event occurs. The same code pattern is used for task and sub-processes with attached border events.

Figure 4.14 shows an example process which waits for a message to arrive




```
  protocol_t protocol;
2 bool protocol_flag;
  void (*protocol_continue)();
4
  timer_t timer;
6 bool timer_flag;
  void (*timer_continue)();
8 long timer_interval = SECONDS(1);

10 void process() {
     timer_continue = &after_timer;
12   timer_flag = true;
     timer_alarm(&timer, timer_interval);
14   protocol_continue = after_receive;
     protocol_flag = true;
16   protocol_receive(&protocol);
   }
18
   void after_timer(){
20   timer_flag = false;
     protocol_stop(&protocol);
22   protocol_flag = false;
     A_start();
24 }

26 void after_receive(){
     protocol_stop(&protocol);
28   protocol_flag = false;
     timer_cancel(&timer);
30   timer_flag = false;
     B_start();
32 }
```

Figure 4.14: Generated code pattern for an event-based gateway which uses a 1 s timeout for receiving a message. We assume an alarm implicitly stops the timer.

within one second. If the message arrives before the timer elapses, the activity `A` is executed, otherwise, activity `B` is executed. The generated code uses the platform API functions for starting a timer (line 13) and a protocol specific reception (line 16) of messages. The code also sets the continuation methods accordingly. If the `after_timer` method is invoked first, it cancels the protocol reception (line 21). We assume that canceling the reception implicitly clears any pending callbacks for the `after_receive` method, which is never executed in this case. The same principle is applied for the opposite case, i.e., if the `after_receive` method is executed first the timer alarm is never executed.

4.4 A Slim BPMN Run-Time

For managing the parallel execution of sub-processes we propose a slim BPMN run-time tailored for WSN. The run-time ensures the proper instantiation of parallel sub-processes. At the same time, the run-time dynamically matches throw and catch events across the process hierarchy. When starting a new sub-process, memory must be allocated for the state variables of the new instance. Conversely, when a sub-process completes, the allocated memory must be freed. Moreover, each sub-process requires a variable-sized queue of event handlers where the delegates for corresponding catch and throw events are registered. Thus, supporting multiple instances requires *dynamic memory management* to

separate the state for each sub-process.

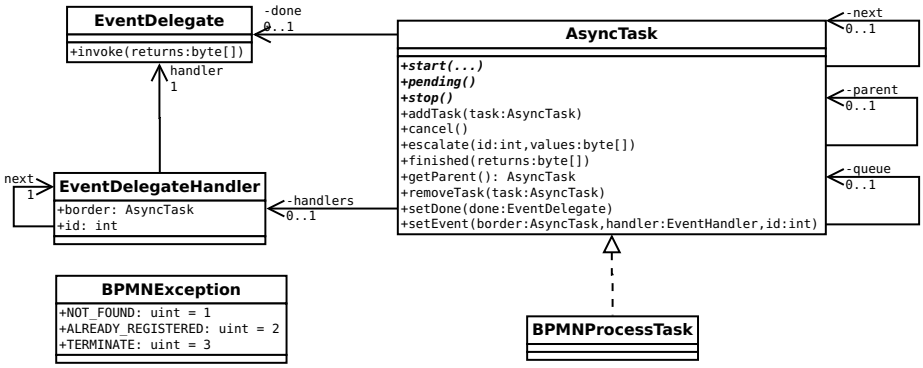


Figure 4.15: UML class diagram showing the abstractions used by the BPMN run-time for managing sub-processes. The `AsyncTask` is abstract and contains common implementations for management functions such as adding new sub-processes, registering events and support for abrupt cancellation. All generated classes extend this abstract class and must implement three methods: `start`, `pending`, `stop`.

In this case, it is natural, to use object-oriented concepts such as classes and encapsulation to describe the behavior of sub-processes. An implementation of the run-time for purely C-based systems such as TinyOS is theoretically possible by using different namespaces and respecting naming conventions. However, memory needs to be pre-allocated to accommodate for the maximum usage required by all sub-processes. Apart from being challenging such an approach can result in an inefficient usage of memory resource. A VM-based system, which provides automatic memory management and garbage collection, such as Mote Runner, is a better choice for the implementation of the run-time. Consequently, in contrast to the previous examples, we use C# syntax for the generated code patterns.

Abstract Functions

Figure 4.15 shows an overview of all the entities and their relationships which are managed by our proposed BPMN run-time. The abstract `AsyncTask` class represent the core the BPMN run-time and implements shared functionality. For each sub-process in the model the compiler generates a class with local instance variables which reflect the current state of the sub-process. Each generated class extends the abstract class `AsyncTask` and must implement three abstract functions: `start`, `pending` and `stop`. While the semantic of these functions is similar to the static case, their implementation accounts for the dynamic behavior.

As before, the `pending` method returns true if no basic asynchronous operations (started by the sub-process) are still active. Again, Boolean flags are used to mark the activity of basic asynchronous operations. The `stop` method stops all basic asynchronous operations which have been started. We assume that the platform API provides for all basic tasks a corresponding stop method with a

similar behavior. Otherwise, the basic asynchronous activities may be stopped by invalidating their callbacks, e.g., using `null` pointers. We assume that the underlying OS will simply ignore invalid callbacks.

As in the static case, each sub-process must implement a `start` method. The difference is that in the dynamic case the `start` method may take parameters which are copied into the local state of the sub-process. The local state is required to separate each sub-process instance. Variables which are used by sub-tasks must be public, whereas variables which are only used locally in the process can be hidden, e.g., marked as private in object oriented code.

Sub-Process Management

instantiation

When a sequence flow in the process model reaches a sub-process, a new instance of the respective class is created. The new instance is then added (`addTask`) to the `queue` of its parent. The queue of instances is entirely managed by the run-time. In addition, the sub-process saves a reference to the parent who started it. The `parent` reference is used for accessing data items in the process model hierarchy. This reference is also required for interrupting event sub-process which must terminate the parent process.

cancellation

The functionality for canceling child sub-processes is shared by the run-time. Cancellation is implemented by the `cancel` method in the `AsyncTask` class. The method recursively calls the virtual `stop` method for all previously started sub-processes. The `cancel` method is typically called by delegates corresponding to events with an interrupting semantic, which abrubtly terminate a sub-process instance.

events

To enable event resolution, the run-time provides a `setEvent` method which saves the reference to a continuation method for each event type which a sub-process may throw. Similar, the `setDone` method saves a `done` reference which points to the continuation method used for the normal completion of a sub-process. The signature of continuation methods is defined by the `EventDelegate` class. For generality, the signature of the continuation method is fixed and allows to pass an array of bytes. Consequently the actual parameters are encoded in the byte array using methods provided by the platform API, similar to the case of persistent variables (cf. Section 4.3.1). While this mechanism is generic it adds some considerable execution overhead.

For handling throw events the run-time provides an `escalate` method which resolves the event handler based on the unique event identifiers. Subsequently, the respective continuation method is invoked. The list of event handlers together with the normal completion handler are automatically managed by the run-time. Algorithm 3 shows the pseudo-code for the event propagation functionality which is implemented in the `AsyncTask` class.

Each continuation method is associated with an event identifier and a corresponding enclosing or `border` sub-process. This association is encapsulated by the `EventDelegateHandler` class. The BPMN run-time matches catch and throw events based on a list of constant identifiers which are generated during compilation. The identifiers become publicity accessible constants for the concrete, generated class which implements the `AsyncTask` class. If the corresponding catch event has an interrupting semantic, i.e., it terminates the task, then the corresponding `cancel` method is invoked for the enclosed `border` or `parent` sub-process. The implementation of interrupting events uses exceptions

Algorithm 3 Propagation of escalation events.

Require: Event *id* to be escalated
Require: Return *values* which are carried by event
  Find handler for event *id* in the list
  if found **then**
    if event handling interrupts a task t, as border event or event-sub-process **then**
      Invoke the cancel method for t
      Remove t from it's parent list of children
    end if
    Invoke the handler function (if valid) with the return *values*
    Destroy the current call stack for interrupting events
  else
    We reached an exceptional state, the event id was NOT_FOUND
  end if

(`BPMNException`) which are not caught and consequently destroy the call stack.

completion

When the execution of a sub-process completes, i.e., the end event is reached, the BPMN run-time calls the method specified by the `done` delegate, if and only if the `pending` method returns true. When a child sub-process completes, either normally or is terminated, it is automatically removed from the queue of its parent (using the `removeTask` method). Algorithm 4 shows the pseudo-code for implementing the completion behavior. This common functionality is again shared through the `AsyncTask` class.

Algorithm 4 Handling completion of sub-processes.

Require: Delegate *done* to be invoked
Require: Return *values* which are carried by the normal completion
Ensure: All children sub-processes and all active operations have completed
  if $\nexists$ pending operations and queue of sub-processes is $\emptyset$ **then**
    if $\exists$ parent process **then**
      Remove ourselves from the parent's queue of sub-processes
    end if
    Invoke *done* with the return *values*
  end if

Example

Figure 4.16 shows a dynamic model, according to the definition in Section 4.1.2. In this case, multiple instances of sub-process A are started in parallel. Thus, the generated code, shown in Figure 4.17, must use the BPMN run-time abstractions described above.

At the parallel split gateway, the three different instances of the sub-process A are started. Each instance may throw an escalation event e. We number these instances (based on their geometric position) using superscripts, i.e., A^1 is the top most instance, A^2 is the middle instance, and A^3 is the lowest instance. The event e is caught on the border of A^1 and terminates this instance. Afterwards, the activity C is executed. For the lower two sub-process, e propagates to

the event sub-process s which handles this event occurrence. In this case, the activity E is executed and the entire process is terminated including all started activities and sub-processes.

In the generated code, the occurrence of e is signaled using the `escalate` method (line 10) at some point during the execution of the main process. The automatically generated constants `E1, E2` (line 3-4) are used for matching the two event handlers. `E1` is only used for the border of A^1, whereas, `E2` is used for handling the propagation of e from both A^2 and A^3. The corresponding events are registered using the `setEvent` method (in lines 26, 32, and 38).

The individual task instances are created (lines 24, 30, and 36) and added to the queue of children for the current process (lines 25, 31, and 37). Prior to starting the new sub-process instances (lines 28, 34, and 40), the continuation methods (lines 27, 33, and 39) for the normal completion are set. The input parameters which are required by A (not displayed in the graphical model) are passed-in as constants.

Depending at which point in time and which instance of A throws e, different cases can occur. If e is never thrown, the tasks B, D, and F are all executed, but C and E are not. Another case is when only A^1 throws e. Now, C, D and F are executed but B nor E are not executed. When only A^2 throws e, before A^1 completed, none of B, D, nor F are executed.

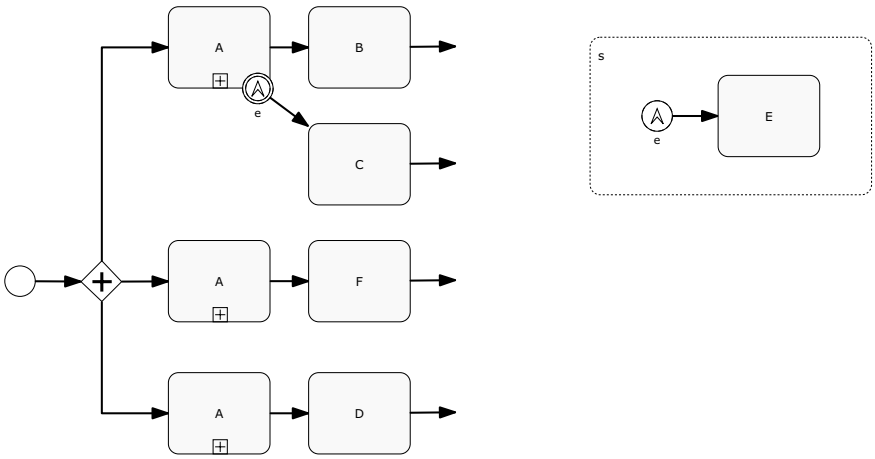


Figure 4.16: A dynamic model where multiple instances of sub-process `A` are started in parallel. The event e, which is thrown inside A is handled either on the border (for the top instance) or using even sub-tasks (for the lower two instances). The normal execution path continues if e never occurs.

```
class A : AsyncTask {

  public const int E1 = 1;
  public const int E2 = 2;

  private int p;

  public void start(int p){
    this.p = p;
    ...
    execution();
  }

  private void execution () {
    ...
    escalate(E1, values);
    ...
  }
}

class Process{

  private void fork () {
    A a1 = new A();
    this.addTask(a1);
    a1.setEvent(a1, event_e1, A.E1);
    a1.setDone(continue_a1);
    a1.start(11);

    A a2 = new A();
    this.addTask(a2);
    a1.setEvent(this, event_e2, A.E2);
    a2.setDone(continue_a2);
    a2.start(22);

    A a3 = new A();
    this.addTask(a3);
    a1.setEvent(this, event_e2, A.E2);
    a2.setDone(continue_a2);
    a2.start(33);
  }

  private void continue_a1 (byte[] returns) {
    B_execute();
    ...
  }

  private void continue_a2 (byte[] returns) {
    C_execute();
    ...
  }

  private void continue_a3 (byte[] returns) {
    F_execute();
    ...
  }

  private void event_e1 (byte[] returns) {
    D_execute();
    ...
  }

  private void event_e2 (byte[] returns) {
    E_execute();
    finished(null);
  }
  ...
}
```

Figure 4.17: Example generated code pattern for the dynamic case with multiple parallel instances corresponding to the model in Figure 4.16.

4.5 Parallelism Detection Algorithm

In general, static code performs better in terms of execution speed and has a lower usage of volatile memory. Consequently it is desirable to generate code in the static mode. However, as we have seen in Section 4.1.2, non-trivial cases can occur where a model requires what we called dynamic behavior. Thus, given a process model, it is desirable to know prior to compilation whether a parallel situation can occur at all during the execution. If no parallel situation can occur, then static code generation can be used safely. Otherwise, the code should be generated using the dynamic mode.

As such, we propose an algorithm which determines whether a model is always static or may exhibit a dynamic behavior. The algorithm performs a pruned exploration of the execution state of a process. For each task t a set of labels $L[t]$ is maintained. Each label is a sequence of numbers, e.g., $\{1, 2, 1\}$, which mimics the execution using tokens (cf. Section 4.1). A split point adds new numbers to a label, and a merge point removes numbers from the label. An overview of the labeling procedure is shown in Algorithm 5.

The same task may appear at two distinct locations in the process, and we will refer to such tasks as being *twins*. During the labeling algorithm, twin tasks will receive distinct names. For example, if a task A appears twice in the process graph, there will be two twin tasks A_1 and A_2. At the end of the labeling procedure, the label sets of twin tasks are checked for possible parallelism.

twin labels

Two labels sequences l^1 and l^2 show that parallel execution of the corresponding tasks is possible if the sequences share a common prefix shorter by one than the minimum sequence length, and they differ in the subsequent position. A mathematical formulation of *parallel labels* is the following:

$$\Pi(l^1, l^2) = \begin{cases} true & \forall 0 < i < min(|l^1|, |l^2|) - 1, l_i^1 = l_i^2 \wedge l_{i+1}^1 \neq l_{i+1}^2 \\ false & \text{otherwise} \end{cases} \quad (4.1)$$

Where $|l^1|$ and $|l^2|$ represent the lengths of the two sequences. For example, the label sequences $l^1 = \{1, 2, 3, 1\}$ and $l^2 = \{1, 1\}$ are parallel, whereas the label sequences $l^3 = \{1, 2\}$ and l^1 are not parallel.

Consequently, two twin tasks A_1 and A_2 may execute in parallel if $\exists l^1 \in L[A_1] \wedge l^2 \in L[A_2]$ such that $\Pi(l^1, l^2) = true$. Moreover, if the label sequence l of a task A is parallel to a previous label $l' \in L[A]$, then A may potentially execute in parallel. The labeling algorithm stops when finding such a task.

The algorithm exhaustively explores all possible execution paths and will thus detect parallelism where in practice it may not occur. The execution possibilities can be limited at run-time for example by the values ranges for decision conditions, certain data dependencies, or due to strict timings of events. In such cases, the user should be able to override this decision and specify that static code should be generated which is more efficient.

Algorithm 5 State exploration using a labeling algorithm to determine whether a process may have multiple instances of the same task executing in parallel.

Initialize the map L of label sets for each task
Begin at the unique start event of the process which gets the label $l \leftarrow \{1\}$
while doing a breadth first search traversal of the process graph **do**
  if current node is a split point **then**
    Split the label according to number of outgoing sequence flows n
    for $i = 1..n$ **do**
      Sequence flow s_i gets a longer label $l' = \{1, \ldots, k, i\}$,
      where the current label of the incoming edge is $l = \{1, \ldots, k\}$
    end for
  end if
  if current node is a join point **then**
    Outgoing sequence flow o gets a shorter label $l' = \{1, \ldots, k\}$,
    where the incoming label is $l = \{1, \ldots, k, j\}$
  end if
  if current node is a task t **then**
    if $\exists\ l' \in L[t]$ such that $\Pi(l, l') =$ true **then**
      The task t potentially has multiple parallel instances
      return
    end if
    Add current label l to the set of labels $L[t]$
    if t is already visited **then**
      while doing a depth first search traversal from t **do**
        Relabel tasks by splitting and merging as before
        if at task p, $\exists\ l_1, l_2 \in L[p]$ such that $\Pi(l_1, l_2) =$ true **then**
          The task p potentially has multiple parallel instances
          return
        end if
      end while
    end if
  end if
end while
for all twin tasks (t_1, t_2) **do**
  if $\exists l_1 \in L[t_1], l_2 \in L[t_2]$ such that $\Pi(l_1, l_2) =$ true **then**
    The twin tasks (t_1, t_2) potentially execute in parallel
    return
  end if
end for

Example

Figure 4.18 shows an example labeling of a process model according to the previously described algorithm. In this case, even though the task `A` appears twice in the diagram, it is never executed in parallel. The labeling shows that in all execution states the two `A` tasks will execute one after the other. However, the labeling algorithm detects that two instances of task `B` (which is highlighted) can potentially execute in parallel.

This case depends on the path chosen at the XOR gateway `X`. Thus, in practice, depending on the conditions at the fork gateway, the parallelism situation may not occur. The reason for multiple instances of `B` is not the parallel AND-split gateway `P`, according to intuition. Instead, the combination of an AND gateway, `P`, and an XOR gateway, `Y`, synchronizes the control flow in an inconsistent manner. This is a case which shows a lack of synchronization, as described in the previous sections, and it hints towards a modeling error.

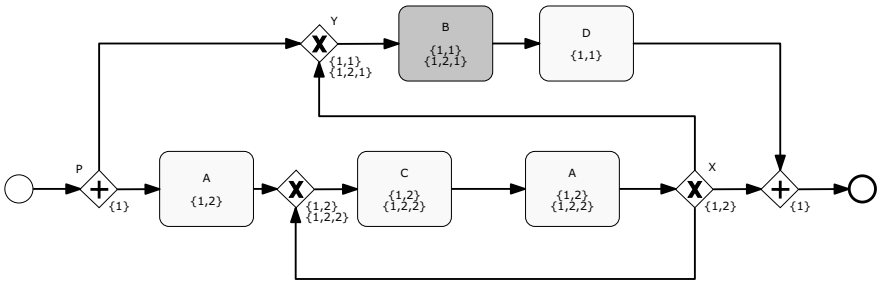


Figure 4.18: Labeling performed by the parallelism detection algorithm. The highlighted task `B` is a candidate for parallelism.

Summary

This chapter focused on the compilation algorithm and patterns used for automatically generating code from process models specified using BPMN diagrams. The challenges of the transformation were analyzed, and different strategies were proposed to deal with the data flow, communication flow, control flow, and events. In addition, dynamic models are supported by means of a slim BPMN run-time. These aspects represent the core of our BPMN to WSN compiler. However, the compiler is but one part in the set of tools necessary for a model-driven methodology. We discuss the full environment required by our methodology in the next chapter.

Chapter 5

Integrated Development Tools

To fully enable the model-driven methodology described in the preceding chapters, we require a set of integrated tools, namely: a model editor, a model compiler, a model debugger, and a model simulation environment. In this chapter, we discuss the functionality of these tools and how they assist the model-driven development of business-aligned applications for WSN using graphical representations of processes. Our description of each tool focuses on the common aspects of BPMN and WSN. Furthermore, we show how these tools interact to provide a comprehensive model-driven methodology.

The WSN development environment forms the foundation for our integrated BPMN tools. The implementation of the integrated BPMN modeling tools is theoretically possible irrespective of the underlying WSN platform and development environment. However, certain features such as an accurate WSN simulation with a debugging interface, integration with back-end infrastructures, built-in maintenance support, visualization, as well as power tracing facilitate the implementation and integration of our BPMN tools. Consequently, we describe the details of the WSN development environment, which provides the platform API, platform compiler, platform debugger, and platform simulator. Our analysis of the WSN development environment uses Mote Runner as an example to extract the set of required features as well as the features which facilitate the integration of our BPMN tools.

5.1 Model Editor

The main tool for modeling business process is a model editor which provides a graphical user interface for users to draw process models. The editor checks the graphical components and only allows connections according to the syntax of the process-modeling language, in our case the BPMN language. Ideally, the editor should be a familiar tool, which business analysts and process engineers use and are trained with.

The editor also provides a uniform and integrated access to the remainder of the modeling and simulation tools through different menus and views. Another functionality of the editor is the ability to manage models, share them with

different users, attach various meta information and comments, keep track of different versions, and maintain history of changes.

Apart from serializing models in different standard XML formats for business processes such as XPDL, most editors also allow graphical models to be exported to common image formats such as PNG or widely used document formats such as PDF. Such an ability fosters the exchange of models between different business participants which do not necessarily have access to a specialized editor or viewer in which they can inspect models.

5.1.1 Existing Editors

The current range of available BPMN editors is extremely diverse. Some solutions are solely drawing programs such as the plug-ins for the popular Microsoft Visio or the open-source Dia. The high-end spectrum of BPMN editors are bundled with BPM solutions and are offered by established middleware providers such as IBM, SAP, and Oracle. However, similarly to the process engine (cf. Section 2.2), the editor is in general tightly coupled with the corresponding BPMS and extensions are often based on proprietary interfaces. In addition to the established middleware providers, there are several solutions, including open-source alternatives, which have attracted a lot of attention both from the research community as well as the industry. The Oryx editor is a popular open-source example which provides not only modeling for BPMN but other graph-based modeling languages such as Petri Nets and EPC diagrams using an extensible Web-application framework. Other promising vendors for BPMS and corresponding BPMN editors at the time of writing are BizAgi, which provides a free BPMN editor but commercial BPMS, Signavio, which offers a commercial product based on the Oryx editor, and the editor from Intalio, which is based on the Eclipse modeling framework.

Figure 5.1 shows an example BPMN editor. A characteristic feature of most BPMN editors is a *toolbox*, shown here on the left-hand side of the editing window. From the toolbox users can drag-and-drop graphical components. The components can be either basic symbols, i.e., without any execution details, or advanced symbols which specify the full execution details. A second feature of BPMN editors is a view over the process model where users can zoom in and out of sub-processes, or hide certain components, e.g., data associations or message flows. This view is shown in the central part of the editor window and allows users to browse models much like a modern Web browser by following the links and references in the process models. For example by clicking on the [+] sign a sub-process can be expanded or opened in a new editor view. In addition, editors provide different context menus for changing the shape and type of graphical symbols or editing non-graphical model properties. Essentially, these properties are Level 3 and Level 4 refinements (cf. Section 1.4) providing the details and data specifications required for execution. In the current example, the properties are shown here in a form-based editing panel in the lower part of the editor view.

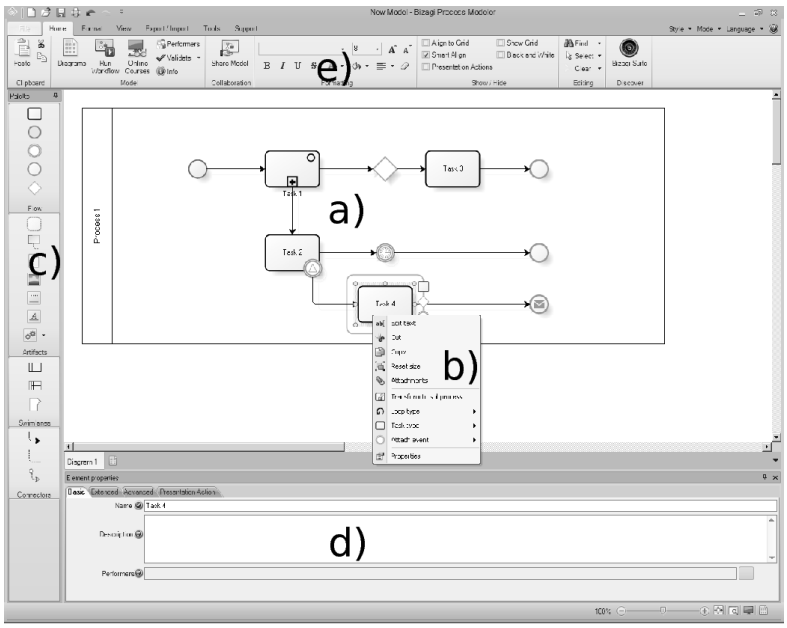


Figure 5.1: A typical BPMN editor (BizAgi Process Modeler) for modeling processes showing a) the process view, b) the context menu for graphical shapes, c) the toolbox of graphical symbols, d) editable non-graphical properties, and e) the common editing toolbar.

5.1.2 Our Extensions

To support the business-aligned development of WSN application according to our methodology, a BPMN editor must be extended to provide compilation and simulation of models. In addition, the editor must provide access to advanced (Level 3) BPMN symbols which corresponding to a WSN platform API. The tight integration between the model editor, model compiler and WSN simulation together with the pre-populated symbols in the toolbox for modeling WSN applications represent the main distinguishing factors which separate our approach from existing BPM solutions and editors.

5.1.3 Importing a Platform API

To fully benefit from a graphical process modeling approach, the API which is available on a certain WSN platform should be visible as predefined components in the model editor. The components can be shown either directly in the editor's toolbox or as part of the shape context menu. The details about execution, data input and outputs as well as references to operations and event types are already populated with default values. Otherwise, the user would have to specify using the non-graphical properties for each model element the operation which should be executed together with the required set of parameters. Such a manual specification is a cumbersome and error prone task. Consequently, the model editor should allow users to import and view the platform API. Business analysts can then easily drag-and-drop the desired WSN functionality into the

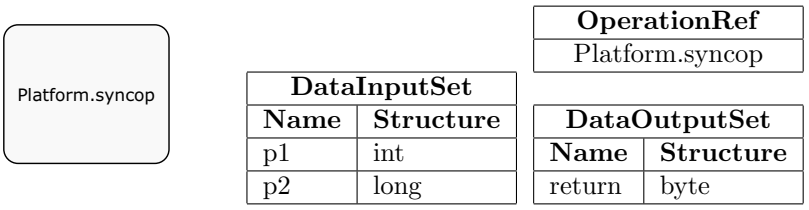
process model.

Synchronous Methods

For synchronous API operations, i.e., which execute to completion (c.f Section 3.2.1), the import transformation is fully automatic. Such operations are transformed to BPMN tasks with pre-populated inputs, outputs, and the operation reference to the platform API method. Figure 5.2 shows the result of such an import transformation for the `syncop` method specified in the `Platform` class. As initial task name the fully qualified method name from the API is used. The task name can be later changed by users to convey meaningful mnemonics for actions. For example "switch light on" is closer to the business language than "PIN.control". To maintain the complete information provided in the model and aid software developers, which might have to deal with the generated source code, these mnemonics are later translated into comments by the compiler.

```
class Platform {
  /// <summary>Some synchronous platform operation.</summary>
  /// <param name="p1">The first parameter.</param>
  /// <param name="p2">The second parameter.</param>
  static byte syncop(int p1, long p2);
}
```

(a)



(b) (c)

Figure 5.2: (a) An example synchronous platform API method. (b) The imported BPMN task together with its (c) non-graphical properties specifying the execution details.

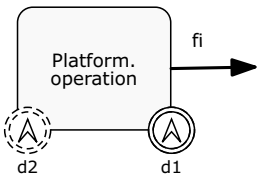
Asynchronous Methods

In contrast to the previous case, importing platform API methods which are asynchronous is semi-automatic due to the different possibilities for expressing events. However, with proper annotations, which can be written as the same time as the API documentation, the import transformation can be fully automated. As described in Section 3.2.1, events can notify about normal completion, exception completion, and significant states.

Asynchronous methods may specify delegate callbacks either at invocation as parameters or beforehand by using special `setCallback` methods. In both cases, a general strategy for importing an asynchronous method to BPMN is to populate the complete `DataInputSet` with all required parameters, including the corresponding delegate. Moreover, delegate callbacks correspond to BPMN events and can be graphically represented.

```
class Platform {
  /// <summary>Some complex platform operation.</summary>
  /// <param name="p1">The first parameter.</param>
  /// <param name="d1" event="SIGN"> Report an intermediate state.</param>
  /// <param name="d2" event="INTE">Signals termination.</param>
  /// <param name="fi" event="DONE">Signals completion.</param>
  /// <returns>Result between [0..9]</returns>
  static int operation(int p1, Delegate d1, Delegate d2, Delegate fi);
}
```

(a)



(b)

DataInputSet	
Name	**Structure**
p1	int
p2	int
d1	Delegate
d2	Delegate
fi	Delegate

OperationRef
Platform.operation

DataOutputSet	
Name	**Structure**
return	int

(c)

Figure 5.3: (a) An example asynchronous platform API method which uses delegate callbacks as parameters for the call. The API method is transformed to a BPMN task (b) with attached border events and corresponding non-graphical properties (c) which specify execution details.

annotations

However for a correct transformation, the semantic of events and corresponding delegates must be specified. For this purpose, we extend the comments, which are provided by the platform API, using attributes as *annotations*. For example, the delegate which marks the normal completion of an operation is specified using the `event="DONE"` annotation, the delegate which marks a significant state but does not interrupt the task is annotated with `event="SIGN"`, whereas the delegate which does interrupt the task is marked by `event="INTE"` and signals an exception completion. If the `event` annotation is not present in the comments of an API method which uses delegates, the import transformation will use interrupting border events by default. If annotations are present, compile-time checks can verify whether the modeled events specify the correct interrupting or non-interrupting semantic. After the transformation, users should be allowed to adjust or correct possible incongruities in the semantic of the API and the corresponding BPMN symbols. In particular users should be able to change interrupting events to non-interrupting events and vice-versa.

An important difference to the synchronous case is that asynchronous operations are typically translated to a group of BPMN symbols namely a task and attached border events. The rationale behind the transformation is to visually encode the semantic of the corresponding events. By contrast, synchronous operations correspond to a single task.

Figure 5.3 shows the first case where the platform API takes several delegates as parameters. The resulting BPMN component provides detailed properties for the `OperationRef` which encodes the full (including class and namespace if required) method name. Additionally, the `DataInputSet` and `DataOutputSet` describe the input respectively output parameters. The second case of asyn-

```
1  class Platform {
       /// <summary>Invoked when ....</summary>
3      /// <param name="d1">Reports an intermediate change.</param>
       static void setCallbackD1(Delegate d1);
5
       /// <summary>Invoked when ....</summary>
7      /// <param name="d2">Signals termination.</param>
       static void setCallbackD2(Delegate d2);
9
       /// <summary>Invoked when ....</summary>
11     /// <param name="f">Signals completion.</param>
       static void setCallbackDone(Delegate fi);
13
       /// <summary>Some complex platform operation.</summary>
15     /// <param name="p1">The first parameter.</param>
       /// <callback name="d1" event="SIGN" func="setCallback1"/>
17     /// <callback name="d2" event="INTE" func="setCallback2"/>
       /// <callback name="fi" event="DONE" func="setDone"/>
19     /// <returns>Result between [0..9]</returns>
       static int operation(int p1);
21 }
```

Figure 5.4: An example asynchronous platform API method which uses `setCallback` methods for signaling various states. The `event` and `func` comment annotations provide the information required for automatic transformation to BPMN. The resulting BPMN graphical components are identical to Figure 5.3, namely a task with two attached border events.

chronous platform API functions decouples the delegate callbacks from a specific methods. The required callbacks must be set using respective `setCallback` methods (lines 4, 8, 12) before executing the respective platform API operation (line 20).

The import transformation can resolve the `setCallback` methods only if appropriate annotations are provided in the source code. In our case, we require that these annotations are part of the source code comments as shown in Figure 5.4. The `func` keyword provides the binding information for events by specifying the corresponding setCallback method. The variable assignments and output result of the imported BPMN task remain the same and are treated identical to the synchronous case.

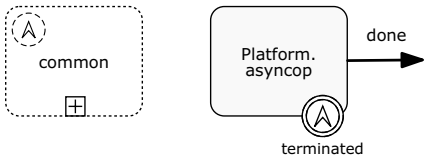
In addition to the generic events discussed above, a platform API also provides basic primitives such as timers, and default communication. These are translated to respective timer events and message events using the corresponding BPMN symbols.

Assignments

For certain event types, e.g., system-wide notifications, it makes sense to use event sub-processes (cf. Section 3.2.1). In this case, the event must specified using a label or string which refers to some model element, e.g., an event subprocess. The same is valid for string references to other model elements such as data objects, which are required for input, output, as well as in conditions. As a general rule, resolution is based on labels and string reference. Thus, variables are resolved based on their names.

To avoid name clashes, the non-graphical `Assignments` BPMN property allows to specify the bindings between labels or strings (representing model

elements) and the parameters required by BPMN components such as tasks, events, or conditions. For parameters which have a unique mapping the model compiler can automatically resolve the desired parameter if the name and type are the same as expected. However, multiple matches, e.g., same type but a different parameter name, must be disambiguated by users using the non-graphical properties.



Assignments	
From	**To**
common	first
common	second
terminated	malfunction
done	finished

Figure 5.5: Example for specifying how events should be bound to delegate parameters by using the `Assignments` BPMN property. The same usage is valid for data object parameters and event sub-processes.

Figure 5.5 shows such an example where events are bound to the delegate parameters. For example the event-sub process `common` handles the callbacks for both the `first` and `second` delegate parameter. The `terminated` event is mapped to the `malfunction` delegate parameter, whereas the `done` sequence is assigned to the `finished` delegate.

5.2 Model Compiler

The model compiler is responsible for transforming business process specifications describing WSN applications into executable code. For business users the compiler is seamlessly integrated into the model editor. The compiler must provide users with feedback with respect to the transformation. Some transformation steps may fail because of inconsistent or incomplete models. Such errors must be visible to users at the same level of abstraction namely in the editor interface where the graphical components which are the root cause are highlighted.

Options

For a versatile solution, the model compiler should support the following main options:

- *platform* — specifies the particular WSN platform for which we compile the process model, i.e., it defines the platform API being used
- *reference(s)* — specifies reference(s) to other platform API or libraries, e.g., which may implement particular communication protocols or special sensor drivers
- *static* — flag which specifies that compilation should follow the static mode, where parallel instances of the same sub-process are not allowed, i.e., without a dynamic BPMN run-time

- *optimize* — flag which enables platform specific optimizations, which recognizes model patterns which can be compacted to a single API call, and generates code which re-utilizes resources such as buffers for communication and timers

Further compiler options describe the input and output parameters. For example, the compiler can take as input either a file which contains a serialized representation for the model or a repository URL where the model can be retrieved from. To allow for inter-operation with tools from different vendors, the compiler should be able to support several standard formats for process models such as XPDL.

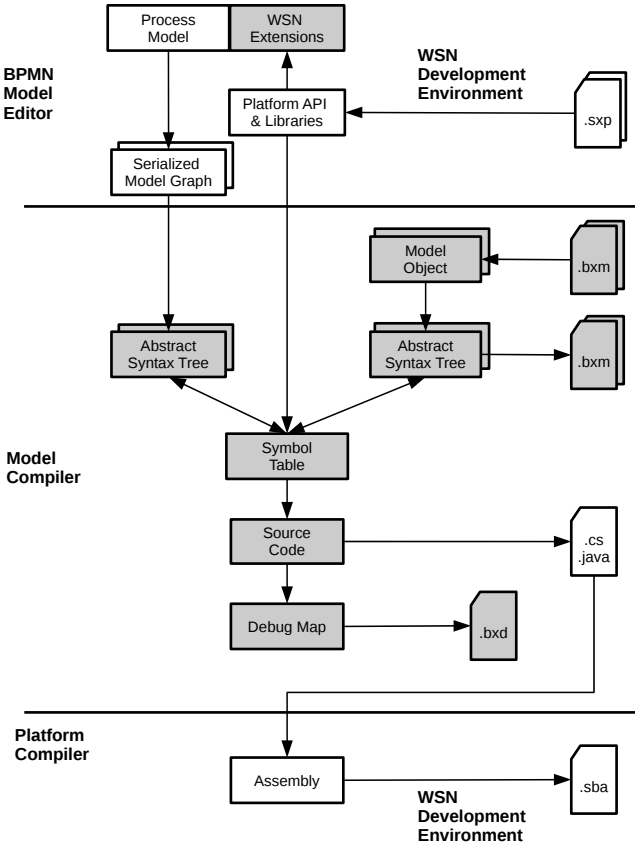


Figure 5.6: The main steps for our BPMN to WSN compiler tool chain showing the flow of artefacts: object model files (bxm), interface descriptions (sxp) for the platform API, the generated platform-specific source code (java, cs), and the model debug mappings (bxd). The platform API is imported from the WSN development environment into the model editor as extensions. The final step in the tool chain invokes the platform compiler to generate the executable binary applications (sba). The contributions of this thesis are highlighted.

Because the compiler generated multiple files, the output must be given as a directory where all generated artefacts are placed. This directory must be also accessible to the rest of the tools such as the WSN simulation environment

and the model debugger. Users working with the graphical interface should not be concerned with any of the files generated by the compiler, nor specifying names or directories. Thus, file names should be automatically generated by the compiler in independent temporary directories. Alternatively all artefacts could be saved in a central BPMS repository, where other business tools can potentially interpret them.

Tool Chain

The model compiler is a tool chain where the final step relies on the particular platform compiler to create the binary form of the WSN application. Figure 5.6 shows the main tool chain steps together with files resulting at each stage, where our contributions are marked by the gray boxes.

The model object (`bxm`) files represent an abstract syntax tree with the information relevant for the compilation, as described in Chapter 4. The `bxm` files can be saved for analysis of the compiler internal structures and operation. If only basic BPMN symbols are used such files are independent of the platform API. Furthermore, the object files can also serve as input, instead of a serialized model.

All object files are then combined to generate the final executable source code (Java or C#) based on the platform specific APIs. For example on the Mote Runner platform the API for libraries (or components) is defined by so called `sxp` files which export the public methods and members of an application assembly. In our case the Mote Runner compiler produces the final executable assembly (`sba`) from the generated source code. For debugging models at the same level of abstraction the required debug symbols are generated by the compiler in `bxd` files.

5.3 Model Debugger

Testing or debugging applications is often viewed as a complementary feature of graphical programming and macro-programming alike. As such, most approaches in the WSN space [114] are not concerned with debugging the generated applications at the abstraction level provided by models. By contrast, our approach provides testing and debugging as an in integral part of the model-driven development cycle. The integration allows to test the behavior of process models early-on and at the same abstraction level. Model-based development of WSN applications thus becomes as agile as test-based development.

Failure Types

In the overall development cycle, there are three different kinds of possible failures. Firstly, *consistency violations* indicate errors in the model which the business analysts have to correct. These errors are reported to the user by the model compiler and enforce the consistency rules defined in Section 4.2.

Secondly, *platform compilation and run-time errors* are the responsibility of the software developers. Either by changing the implementation of the platform API or by adding additional constraints, they have to ensure that a model which passes all consistency checks does not cause such failures. This is important because a common business analyst cannot and should not handle compiler

error messages and stack traces. Software developers can use the platform tools (platform compiler, platform debugger, platform profilers, oscilloscopes, etc.) to test and resolve such issues.

Last, *exceptions* can occur as part of a valid process execution and are represented as events in BPMN models where they can be caught and handled. In general, we recommend to avoid the usage of exceptional events in higher level models because they introduce a non-sequential control flow which is sometimes difficult to handle for people with little background in IT.

Role of Model Debugger

Exception in a process should be inspected and tested using a model debugger which represents a high level-view over the application. The model debugger helps business users in finding problems in the application specification at the process model level. For example, an event fires too early, or a task never completes because the timeout is too short. In a sense, a model debugger is similar to the source code level debugger for a C program. By analogy, the platform level debugger is akin to the disassembly of the same C program.

The process model debugger could be an extra view in the model editor or provide its own graphical user interface. For testing to be effective, the model debugger should be integrated with the model editor and model compiler tools. For ease of use, both tools should be based on the same graphical user interface thus providing common menus and a coherent look and feel. Such an integration reduces the development cycle from making changes to executing a running application.

Debug Map

To support the debugging of business process the model compiler creates a mapping between the graphical elements and the generated source code. By construction, for each graphical element possible multiple code sections are generated. Thus the generated code section are mutually exclusive. Consequently, there is a unique mapping from each code section to a single graphical model element. In addition, we assume all graphical model elements have a unique identifier which is managed by the model editor. Whereas, the code sections are uniquely identified by the file name and start&end line numbers. Because our compiler generates only a single file, it is sufficient to keep a list of graphical identifiers and the start&end lines of the code sections which are generated. Thus, when a simulation is paused or stopped because of a break condition, we can resolve and highlight the current graphical element which is executed. In this case, the current execution line is queried from the simulation and the graphical element is selected whose corresponding code section includes the current execution line.

The debug map is generated in the last step of the compilation while outputting the actual lines for the platform specific source code. For this purpose the abstract syntax tree (see Section 4.2.2) keeps track of the complete properties of graphical BPMN elements. The debug map is read when setting up the simulation for a WSN application and before loading the executable application. When a source code break-point is hit in the simulation, the debug map is used to uniquely resolve the corresponding graphical element. We assume that the

mote on which the break-point happened as well as the source code line can be queried from the simulation. When a break-point is set in the graphical model, possibly multiple break-points are also set in the simulation. Break-points are set on all start lines of the code sections corresponding to the graphical element. For better control, break-points can be set either on all or only on a user selection of the motes which execute the corresponding models.

5.4 The WSN Development Environment

The WSN development environment provides the complementary features required by our BPMN tools. In this section, we describe in detail the features which are strictly necessary for our tools as well as the features which facilitate the implementation and integration of our methodology. In contrast to the purely BPMN tools, the WSN development environment mainly addresses software developers rather than business analysts.

From a high-level perspective, our model-driven methodology requires at the very minimum a platform API which defines a particular WSN and thus the BPMN symbols which modelers can use (cf. Section 5.1.3). Moreover, a matching platform compiler is required to create the executable binary application from the code generated by the model compiler. Our model debugger does not rely on existing business process simulators as we want to test the behavior of the actual application which will be deployed and not simply test the graphical specification. Consequently, we use the underlying WSN simulator which must execute the same binary application as the one generated from the model.

We assume that integration, visualization, deployment, and management of applications is already handled in an efficient and coherent manner by the WSN development environment. Moreover, the implementation of the platform API or other software artefacts, potentially created using classical implementation methods, might require testing and debugging which can be achieved using a platform debugger. Such an extensive set of functionality is already provided by the WSN development environment and does not represent a main contribution of our thesis.

Our analysis of necessary and facilitating features uses Mote Runner[1] as an example for a WSN development environment. However, the process modeling and code generation concepts presented in this thesis are applicable to other WSN platforms with corresponding development environments such as the Contiki platform and the respective COOJA simulator [132].

5.4.1 Mote Runner

Mote Runner is a coherent development environment for WSN applications which integrates an extensible and accurate simulation. We highlight the features which help address the previously introduced WSN application development challenges.

Figure 5.7 shows an overview of the Mote Runner environment which consists of the following components: i) a simulation infrastructure for motes, ii) the operating system and libraries running on both virtual and physical motes

[1]The Mote Runner environment represents a collaborative effort of the entire team at the IBM Research - Zürich Laboratory. This section have been submitted for publication.

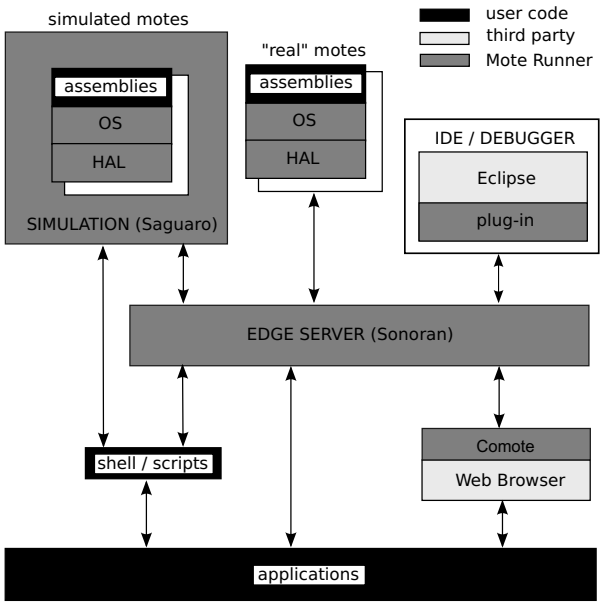


Figure 5.7: Overview of components in a Mote Runner development and deployment environment. The only software difference between a development simulation and a physical deployment is the "hardware" part of a "real" mote. Application code for on-mote processing, control scripts on edge servers, and visualization remain unchanged.

which form the run-time for on-mote assemblies, iii) a middleware interface for applications which integrates both physical and simulated motes, iv) various clients for scripting, debugging and Web-based visualization. In the following, we discuss the individual components and how they help development, maintenance, simulation, debugging, and integration of WSN applications.

One important aspect is to allow developers to use standardized and familiar high-level languages such as Java and C# to program WSN applications. We structure application code which executes on a mote into an *assembly* which can be dynamically loaded and reused as a library much like a JAR file. Multiple assemblies can execute on a mote at the same time and they represent portable binaries. For rapid development of application components in the backend, we enable developers to use JavaScript for control scripts and Web-based visualization.

VM-Based Run-Time Environment

On both simulated and physical motes the same operating system (OS) provides an execution run-time for applications. The Mote Runner OS is designed for small-embedded systems, primarily targeted at motes of a sensor network. Although programmable in Java and C#, its run-time environment for these languages is different from what developers are used to on conventional PCs.

architecture

The Mote Runner OS follows a layered architecture and assumes a hardware

platform providing an 8-32-bit CPU with a minimum of 8 kB of RAM and 64 kB of flash or EEPROM, a power source, wireless and/or wired communication capabilities, and optionally some sensors and/or actuators to interact with the physical world. While the Mote Runner firmware runs on off-the-shelf motes such as MEMSIC's IRIS, its operating systems and higher-level services are platform independent. All hardware-specific plus some performance-sensitive code is encapsulated in the hardware abstraction layer (HAL).

One of the most interesting features of the Mote Runner OS is the tight integration with an embedded virtual machine (VM) [22]. By introducing a VM, applications become run-time portable in the sense that the same binary can be executed on different hardware platforms with no recompilation required[2]. Specific features of individual motes can still be made available via optional interfaces while their presence or absence can be listed in a profile describing a mote's configuration at a given point in time. Application development then effectively becomes independent from individual mote hardware configurations, being more tied to a mote's feature set and potentially its non-functional characteristics instead. This abstraction maps directly to the platform API as described in Section 3.2.2.

efficiency considerations

The Mote Runner VM follows a stack-based architecture, which offers a more compact application code representation as opposed to register-based VMs [154]. To support a low-power operation, the VM allows the embedded device to go into deep sleep for as long as possible, only being active or communicating when necessary (e.g., in response to certain events). This is in line with the underlying reactive programming model of a WSN (cf. Section 3.2.1).

For an efficient and compact implementation able to execute on a very small and, hence, commercially attractive hardware platforms, the Mote Runner VM byte code is statically typed, which saves additional lookups to determine the type of the objects on the stack. Having a typed byte code makes it possible to map various statically typed high-level programming languages onto the byte code, most prominently Java and C#. We focus on statically typed high-level languages as they allow code analysis and verification, and static type flow analysis to permit optimizations before code is loaded into a WSN to further save on-mote resources including memory and power. The Mote Runner compiler tool-chain finally maps high-level programming languages to optimized byte-codes [22] specific to our VM. This represents the platform specific compiler which is the last step for the model compiler (cf. Figure 5.6).

other VM approaches

Using a VM and high-level programming languages such as Java for developing WSN application has been extensively studied by the research community. An overview of current virtual machines for WSN is given by [27]. Most solutions are specialized on a particular application or abstraction. A prime example for such VM specialization is SwissQM [117] which provides a query abstraction for an entire network. One of the first attempts of application-specific virtual machines is Maté [91] running on top of TinyOS. The Maté VM introduces tailored byte codes to compress the representation of a sensor application with the main goal of enabling dynamic uploads. Generic VM approaches are based on the J2ME standard and require significant hardware resources, like the Squawk VM [157] operating SunSPOTs using a 32-bit ARM. The Mote Runner multi-

[2]The hardware-independent part of the Mote Runner OS, in contrast, is compile-time portable in the sense that the OS has to be recompiled for each hardware platform.

language VM focuses on energy-efficiency and targets more limited 8 and 16-bit platforms because larger hardware platforms inherently consume more energy [142]. Energy consumption is not the main focus of more recent proposals for Java VMs for motes such as Darjeeling [17] and TakaTuka [4].

5.4.2 Simulation Environment

The WSN simulation environment is responsible for the accurate simulation of sensor networks in terms of both communication, resource usage, as well as execution. Several such systems exists which are either simulating generic network communication like the ns2 [172] or are specifically tailored at WSN. Examples include the extension for ns2 presented in [118], the TOSSIM [92] environment for TinyOS networks, the Avrora [166] simulator for specific AVR-based MCUs, or the COOJA simulator [132] for the Contiki OS.

An important requirement for a simulation environment is to closely match the hardware deployment. This match must be an accurate estimate in terms of energy consumption, execution time, message transmissions (including collisions), packet error rate, and clock drift. These are necessary features for our BPMN model-driven development. Only some of the available simulators are able to fulfil these criteria. In general, the simulators are tied to a particular run-time environment and to a particular hardware.

available tools and their limitations

Even though simulation environments and various tools for debugging exist, we quickly reached their limitations because the application we were simulating was not the same as the one deployed on the real hardware. Moreover, we could not stop and inspect the state and source-code for the entire distributed system. In addition the simulation was not able to accurately model the wireless link behavior we observed in the real environment. For example, using TinyOS and TOSSIM [92], nodes must be simulated as MicaZ. When accessing more specific radio functions only available at the hardware abstraction layer (HAL), the simulation did not cover the same code as the deployment system and thus was misleading when debugging an application. Even if individually the existing set of tools [18, 37, 155, 68, 166] are sophisticated and are able to achieve excellent results they are often very difficult to integrate and use in a unified manner for developing WSN applications. Maintaining a diverse set of tools and their environment also requires an amount of time and expertise that should not be underestimated.

simulation engine

Saguaro is the simulation environment for motes running the Mote Runner OS/VM. In fact, complete WSNs of arbitrary topologies can be simulated with all motes running within a single process. Each simulated mote hereby executes precisely the same OS/VM with the same memory image as hardware motes but only relies on a different HAL that integrates with the simulation environment. In addition, features such as power consumption, packet error rate, CPU speed, and clock drift are simulated.

The transmission of radio messages is also simulated and precise at the nanosecond level. The simulation takes both mote location and signal quality into account based on a basic model of quadratic decay in signal quality vs. distance. The model for the wireless channel can be refined by external components, though, taking both structural and environmental conditions into account; for this, the simulation provides suitable hooks already. It is further possible to *pause and continue* a simulated network as a whole for inspection,

enable logging and tracing of systems events at various levels of verbosity, and to run the simulation in a *fast-forward* mode during which all sleep periods take no real time. An external sensor-feeding framework finally allows providing both deterministic and randomized test data for regression and stress testing sensor-network applications. The Mote Runner simulation is also instrumental in providing source-level debugging facilities for assemblies running on simulated motes.

flexible break-points

To facilitate debugging the simulation environment offers *break-points* not only on source code lines but also on events controlling the behavior of motes. For example the simulation allows to stop when a radio message is received by a mote's radio module, or when the VM is active as well as when it exits, when a new assembly is loaded or deleted. All break-points can be individually controlled or globally set according to the developer's debugging focus.

built-in power tracing

Another feature integrated into the simulation environment is *power tracing*. Based on a mote's configuration, the simulation records changes in the drawn current in nA. The consumption trace can be used by scripts and visualization tools to keep detailed power consumption statistics and plot instantaneous graphs. This feature is required to ensure an energy efficient execution through continuous monitoring.

5.4.3 Platform Debugger

The platform debugger is equivalent to a disassembly tool meant for technical users and software developers. The platform debugger helps users find the cause of problems in the implementation of the platform API or software libraries. Such a tool is essential for testing the software building blocks used, as higher level abstractions, in process models.

familiar IDE

To debug at the source code level, Mote Runner provides several plug-ins which extend the widely-used Eclipse IDE. Eclipse itself provides a state-of-the-art familiar look&feel to software developers which includes code completion, refactoring, code searching and navigation, and can be extended with plug-ins such as version control clients.

Although Eclipse primarily provides a Java environment, the Mote Runner IDE supports both Java and C# for assembly development. Using standard high-level languages such as Java for programming WSN application allows even mainstream developers to adapt quicker to such projects. Mote Runner assemblies can be used like JAR files, which allows creating libraries offering specialized functionality. Furthermore, the assemblies can be exchanged and even commercialized.

WSN-tailored features

Aside from integrating C# support, the Mote Runner IDE adds three features to the standard Eclipse installation. Firstly, the Eclipse build component has been extended to seamlessly integrate the Mote Runner command-line tools. Compiling a Mote Runner assembly invokes the Mote Runner compiler tool chain and any error messages are shown in the Eclipse error log.

Secondly, the Mote Runner IDE supports source-level debugging against the Mote Runner simulation environment. All the usual features of source-level debugging such as setting breakpoints, inspecting variables, and single stepping are available.

Thirdly, to accommodate the distributed nature of sensor networks, whenever the simulation is paused, it is possible to inspect the state of all motes and

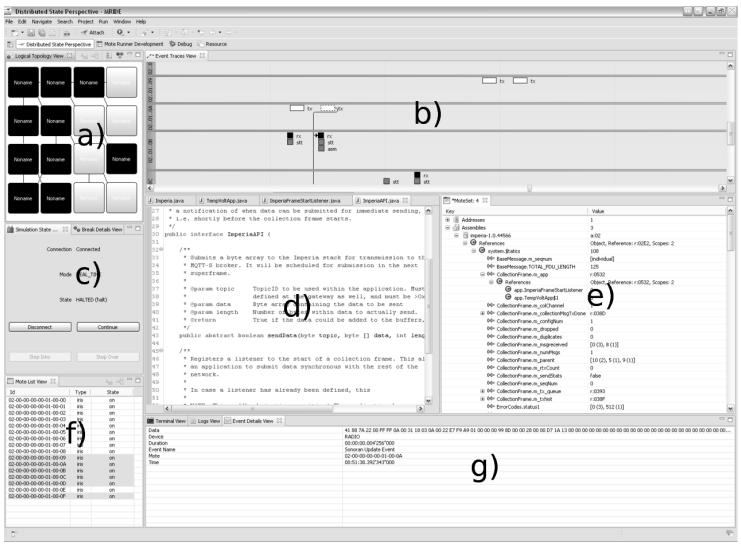


Figure 5.8: The Mote Runner distributed and source-level perspectives integrated into the Eclipse IDE: a) logical network connectivity, b) time-line of events depicting the exact point in time and duration for radio transmissions and receptions, log output, sensor sampling, c) simulation control, d) step-by-step source code debugging, e) content of current variables on a single or multiple motes, f) list of nodes and their state, g) event details, simulation console, and filtered log output.

a time-line trace of all log messages, including the network frames exchanged up to that point. We refer to this feature as *distributed debugging*, which is shown in Figure 5.8. The message trace is particularly helpful when debugging network protocols which require time synchronization.

common debugger interface

The Mote Runner IDE, which represents the platform debugger, and our BPMN model debugger, use the same interface to the simulation. The interface allows to set and query breakpoints and the complete state of the motes in the simulation. In the case of the BPMN debugger an additional mapping is required to resolve the graphical elements. During the development of our integrated set of tools we employed the platform debugger for inspecting the behavior of the generated code.

5.4.4 Integration and Visualization

Wireless sensor networks are not isolated, and the data they provide is propagated to more powerful backend systems for integration with existing business process, visualization, analysis, monitoring, and persistent storage. Most sensor networks have to be connected to some larger backend infrastructure, at least temporarily, to transfer sensor data gathered or receive new settings. Backend connectivity is achieved through one or more edge motes wired to an edge server (cf. Figure 5.7). The edge server itself may be a single machine or a network of machines. The edge motes can be either physical or simulated and the middleware integrates communication using the same API.

Middleware

Sonoran is the off-mote (i.e., running on a server) JavaScript-based programming, management and integration middleware for Mote Runner. The middleware provides and maintains the connection to the edge motes, and implements a broad set of commands to interact with any connected edge mote in different ways from shell commands to scripts to external applications. Using JavaScript as the middleware language has the advantage of being able to port large parts of the integration and visualization code to a Web-based interface. Such an approach is in-line with the current Web of Things approaches [59] which aim to Web-enable sensors using RESTful services.

scripting facilities

Internally, Sonoran uses the V8 JavaScript engine [56], which is extended by a number of native functions (e.g., for file and network I/O) to service the Sonoran JavaScript framework. On top, a number of interfaces exist allowing the interaction with Sonoran and motes. Sonoran further features a command-line shell, and an HTTP interface to its built-in Web server to setup and communicate with a network of motes. Additionally, Sonoran can execute user-defined scripts and applications building upon the base Sonoran functionality.

In practice, Sonoran represents the interaction point for most third party components and Web applications as well as for interactive tools such as the Mote Runner shell. This shell provides functions to create and manage simulation processes, create simulated motes or to connect to hardware motes, exchange messages, and execute management protocols such as loading and deleting mote applications.

back-end integration

Events generated by both simulated and physical motes can be processed in a consistent way and further propagated to back-end systems. Typically integration with third party applications is realized using a standard socket interface. Examples for successful backend integration are solutions working with the open publish/subscribe MQTT-S [167] standard and the event generation mechanism in the IBM Tivoli Netcool/OMNIbus management platform.

For non time-critical applications, it might be desirable to combine within the same scripting environment both simulated and real motes. Due to the common API and event processing mechanism, the simulated-hardware combination is possible using the Sonoran middleware.

Visualization

Comote is a JavaScript library that supports the development of Mote Runner Web applications. It connects to Sonoran to give access to individual motes, the global assembly cache, radio messages, and console log messages. Furthermore, Comote provides a terminal abstraction to issue shell commands and collect their responses, and a socket abstraction to exchange data with individual motes.

Like most JavaScript libraries, Comote is purely event-driven. That is, an application registers event handlers with Comote services such as its mote registry to be informed whenever the state of the wireless sensor networks has changed (e.g., a new mote has been created) or some event has happened (e.g., a radio message has been exchanged).

In addition, Comote allows applications to interact with motes in a sensor network directly. For these interactions the concept of sockets has been implemented with usage patterns resembling those of sockets typically found on

desktop operating systems (e.g., create, bind, use, close).

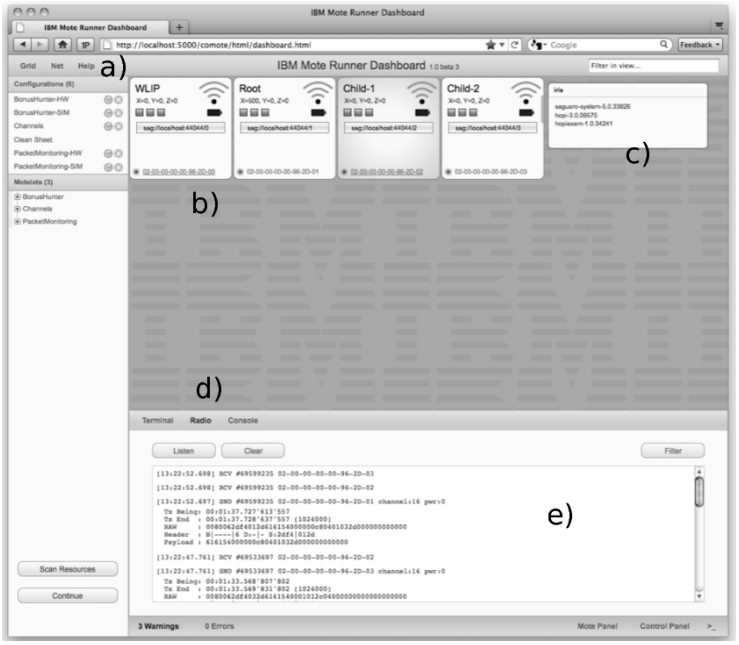


Figure 5.9: Web-based Mote Runner dashboard used for management and visualization of WSN applications showing the a) tab to switch between different views, b) grid view with current status, battery level and unique addresses (EUI) of motes, c) list of installed assemblies for the selected mote, d) tab selector for control panel functions, e) exchanged radio messages on all channels with time-stamps and addresses.

dashboard application

The most often used Web application based on Comote is the Mote Runner dashboard, shown in Figure 5.9, a graphical interface to monitor and interact with sensor networks running Mote Runner. Out of the box, the Mote Runner dashboard provides two different views on a given sensor network. Firstly, a so-called grid view that shows the basic parameters of motes within a network such as their address, location, battery status, or which assemblies are loaded. Secondly, a net view that shows the spatial distribution of the motes within a network including their respective communication ranges.

For interaction with the sensor networks, the dashboard further provides two panels, namely a mote panel and a control panel. The mote panel allows new simulated motes to be created, to connect to physical wired motes, to inspect and update the state of motes, and assemblies to be loaded or removed. The control panel, in turn, provides access to a Sonoran terminal, a radio console for tracing radio messages exchanged, and an error console. Furthermore, the views in the dashboard are customizable for different applications. Such extensions use HTML and the Comote library, namely sockets, to send and receive data to and from an edge mote connected to a potentially remote PC.

5.4.5 Deployment and Maintenance

The *mote manager* is a core feature of the Mote Runner OS/VM which implements a portable mote-level maintenance functionality. The mote manager is a built-in assembly which allows loading and deleting assemblies, as well as querying mote information. For example, the list of installed assemblies, the stack and heap resource usage, the battery status, or the version of the OS and firmware can be queried before deciding for an upgrade. Moreover, through the mote manager, a mote's persistent unique address (EUI) can be updated dynamically.

built-in version information

Keeping track of different version of applications which are deployed or still under development is a tedious task. As such the Mote Runner compiler tool-chain provides a unified built-in support for version information using major and minor numbers. For example, an application assembly always starts at version 1.0. Changes to the public API of an assembly will force an increment of the respective major and minor numbers. If the public API doesn't change, it is possible to recompile the assembly using the same version number. Once the public API changes, the compiler distinguishes between changes which maintain compatibility and those that break it. If features are added, such as methods, field, classes, interfaces, the new API is backward compatible and the minor version is increased. If features are changed then compatibility is broken which requires increasing the major version.

The tool-chain automatically keeps track of these changes by comparing the newly compiled assembly against the present export (sxp) files. In addition, prior to loading an assembly, its dependencies on other assemblies are checked by the mote manager using the version information. Consequently, a mote will reject assemblies with unmet prerequisites. Thus, the model compiler keeps track of the various Mote Runner versions for the generated assemblies and maps this to different model versions. Such information is valuable in case of application updates and recalls.

over-the-air programming

Over-the-air maintenance is an essential feature for wireless sensor networks. As such, the Mote Runner OS has a built-in management protocol for wireless motes (WLIP). The protocol transparently routes all messages to the mote manager which is able to perform all maintenance functions as described above. Moreover, the WLIP operation mode can be programaticaly enabled and disabled by the application.

For example, when the top-level process which describes a WSN application terminates, the corresponding group of motes can switch to the management mode. This behavior simply assigns a meaning to the end event in top level processes. In this way, new instances of the process can be installed on motes. Afterwards, the motes can be restarted for the new process to begin operation.

Summary

We briefly summarized the set of *necessary* and *facilitating* features the WSN development environment should provide. These enable the implementation and integration of BPMN modeling tools which are able to accommodate physical devices into business processes.

necessary features

In terms of WSN debugging, break-points, which are activated on events and on source code lines, are a necessary feature for the WSN simulation. Further-

more, the WSN simulation must be accurate in terms of execution, radio communication, power consumption, packet-error rate, CPU speed, and clock drift to provide estimates which are as close as possible to real deployments. Sensor feeding is another required feature which helps match a real environment based on models for data readings. For the platform debugger, a necessary function is the source-code level inspection for both the platform API and the generated code. Moreover, the platform debugger should support the distributed inspection of the simulation state for the entire WSN. In addition, we deem integration, visualization, deployment, and maintenance, including versioning information, to be necessary features of a WSN development environment. This last set of features provides full control and an overview of the behavior of both simulated and real processes.

facilitating features

Some of the features presented in the preceding sections we deem to be facilitating, i.e., they ease the integration of WSN with the BPMN tools. One facilitating feature is the possibility to pause and fast-forward the simulation at any point to inspect the state. This enables finer control over the simulation of processes. A facilitating feature is also the integration of the platform debugger into a familiar IDE. This enables the platform tools to be used by a larger range of software developers, not only specialized developers for embedded systems.

5.5 Process-Driven Development for WSN

The development process using our model-driven approach spans over several stages which are integrated using the previously introduced set of tools. Figure 5.10 shows which of the tools is used at each stage. In the following, we discuss in detail the steps performed at each stage.

In the classical BPM cycle (cf. Figure 1.1), processes can be simulated during the modeling phase. However, the simulation is based only on the graphical model, which is usually annotated with statistical information [79, 169, 184] about the time required to perform each activity and each transition. In our model-driven approach, testing represents a separate stage. Moreover, in our case, simulation and testing are based on the actual implementation.

5.5.1 Modeling

For business users the BPMN model editor represents the main tool during the process *modeling* phase. The editor is the starting point for designing and developing process models. Here the business consultants and managers will first design a high-level process which corresponds to the Level 1 strategic view (cf. Figure 2.9). This view may already include some of the core entities in the WSN, which are represented as pools.

The high-level process serves as the basis for business analysts which will refine the process by providing the Level 2 operational details. The refinements add messages, events, and exception paths to the high-level design. At this stage, the BPMN graphical symbols do not necessarily contain execution details. A more specific refinement would use the WSN toolbox which provides reusable building blocks.

Several iterations may be required until the strategic and operational details reflect the desired process. To fully describe a complex process, external business

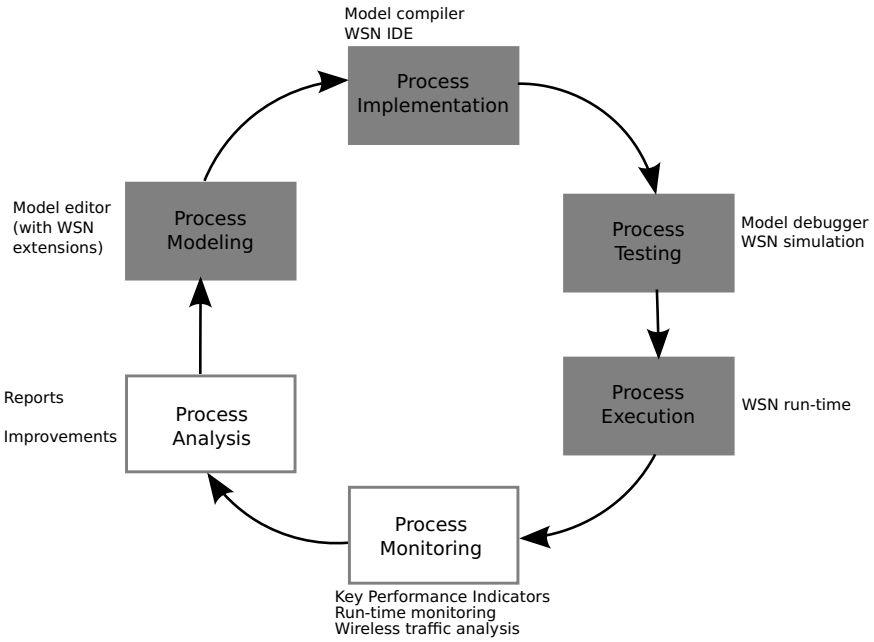


Figure 5.10: Overview of tools and their roles in the BPM cycle. The stages where our integrated approach enhance BPMN solutions with WSN functionality are highlighted.

partners may be involved. The model editor helps all participants in managing this iterative approach by keeping track of different versions and changes. The WSN extensions facilitates the process design by providing an overview of what components are readily available and which functionality requires a custom implementation.

5.5.2 Implementation

At this point, process designers can already run the model compiler. Essentially compilation replaces the manual *implementation* phase which is required in the classical software development of a business process (cf. Section 1.4). A Level 2 (operational) business processes may already be compiled to an executable application. The prerequisite for the compilation is a model which only uses basic symbols, such as timers events and raw communication primitives. As described previously, these are mapped to the default timing and communication functions which are assumed to be available on all WSN platforms.

For compilation to succeed, a model must only use elements which are directly linked to an underlying platform API (cf. Section 5.1.3). This is the case when the appropriate graphical elements from the WSN toolbox are used. The compiler will check all consistency requirements (cf. Section 4.2) are met and will inform the modeler with appropriate error messages. The compilation of a model must be iterated until no compilation errors are reported.

technical refinement

If the BPMN symbols do not yet provide the execution semantics, then an additional *refinement* step is required. During refinement, a business user with a

technical background can specify the required BPMN properties and execution details (Level 3). The model compiler facilitates this step by providing error messages which help identify and resolve common problems such as the wrong set of parameters or using the incorrect names for platform API operations. At this stage the format for the message payload and data handling must be specified.

Such a refinement is similar to a manual software implementation but with a greatly reduced complexity because choices are limited to the available graphical elements. The toolbox providing the WSN platform API serves as a reference and documentation. Alternatively, some low-level components which are not yet provided by the platform API may require implementation using traditional software development methods. Such specific components can be included as embedded scripts tasks or as reusable libraries. The latter can be imported into the model editor. We favor the second variant as functionality can be reused across different models.

Ensuring that the semantics of business activities and events in the model matches the refinement requires the expertise of both the technical and the business. Thus, several iterations and cross-domain discussions may be necessary to reach consensus, as in the previous phase. The resulting model may differ with respect to the initial high-level design. Because execution refinements are explored at this stage, the decision to support static or dynamic processes (cf. Section 4.1.2) should be taken.

5.5.3 Testing

After the business process model successfully compiles to an executable application, users can move to the process *testing* phase to validate the behavior of the process in conditions similar to a real environment. Business users test the behavior of models at the same level of abstraction by literally stepping over actions and observing the changes in the main sequence flow due to events occurring during the course of the process.

For this purpose, the WSN simulation environment must be started and controlled from the process development environment, in our case the BPMN editor. To specify the network configuration and various simulation parameters, we use a conversation diagram model. Such models specify the desired network topology, the participants, and their cardinality (see Section 3.2.3). If no simulation configuration is specified, default settings are used for the simulation, namely one instance for each participant and a topology where every WSN node can reach every other node. Further configuration settings for the simulation, such as packet error rate or clock drift, can be specified in detail by users with a technical background using (Sonoran) scripts.

In certain cases, testing the process model using the model debugger reveals results which the business users cannot explain or which do not match the specification of the platform API. Such cases should be rare in practice. However, if they do occur, they hint towards a problem in the implementation of some parts of the platform API which is used by the model. Consequently, a platform debugger is required to identify and amended such problems. This expertise falls onto the software developers who are familiar with the corresponding platform. After such issues are fixed, the entire process can be simulated again by business users to verify the desired behavior. If the platform API suffered changes, it

may be necessary to re-import (cf. Section 5.1.3) the API into the model editor.

5.5.4 Execution

The last step is the process *execution* where the application is deployed onto real hardware. For deployment, we rely on the support of the underlying runtime support for dynamic application management together with the version information. Ideally, upgrading to a new solution does not require physical access to the motes and application upgrades can be pushed over-the-air.

Now the process executes and it can be monitored and analyzed. The analysis may reveal improvements to the process based on KPIs. The consequence may be that a new process model which accommodates for the improvements or new requirements is required. Thus, the development cycle is closed and a new process iteration starts from the modeling phase.

5.6 Implementation Considerations

In this section, we discuss the considerations for our proof of concept implementation of the integrated modeling tools. Our implementation supports all previously described requirements but for minor functional and usability limitations which do not affect the general methodology. The current implementation allows the description of the behavior of a mote or group of motes including testing and simulation with debugging support. However, the current implementation does not allow changing data object values in a stopped simulation. Furthermore, injecting external events is not integrated into the model editor but has to be done by means of external tools, either Web-based, such as the Mote Runner dashboard or command line tools, such as the Mote Runner shell.

5.6.1 Web-based Editor

Figure 5.11 shows our model editor which is a Web application based on Oryx [32], a BPMN 2.0 editor distributed under the MIT license. The advantage of a Web-based solution is that virtually no installation overhead is required to start modeling, sharing, and testing WSN applications integrated with business processes. Users simply point their browser to the URL of the model editor, which runs on a server machine. Users need not maintain a diverse set of tools. Moreover, updates to building blocks and the platform API can be pushed to the server and are thus immediately visible for all models.

We extended the functionality of the Oryx editor to include plug-ins that integrate the WSN platform API, the model compiler, and the WSN simulation environment for testing and debugging process. From a technical perspective, the model compiler and the WSN simulation environment are part of the back-end of our integrated BPMN tools for WSN. The plug-ins enable multiple, concurrent[3] users to model applications in parallel. In the back-end, the compiler plug-in calls the model compiler which generates C# or Java source code for the Mote Runner platform. The testing and debugging plug-in creates

[3]While users can model applications in parallel, the current implementation does not support concurrent debugging. In other words, only one user can debug a WSN at a time.

the corresponding sensor network configuration for the Mote Runner simulation according to conversation models, i.e., BPMN pools and their respective cardinality.

The communication interface between the Web browser, our model compiler tool-chain, and the sensor network simulation framework uses a RESTful API with data encapsulated in JSON (JavaScript Object Notation) messages. This is a natural choice because the browser uses JavaScript and JSON to display the graphical interface elements. Furthermore, the interface which enables interaction with the simulation uses the same encoding. In addition, the Oryx editor can export models to a proprietary JSON format. For a portable solution, models can also be exported to standard XML formats such as XPDL. Thus, our implementation can be extended to accept different formats as input.

5.6.2 Debugging View

The editor window in Figure 5.11 shows a process model which is currently tested and debugged. Break-points are integrated as overlays to model components which can be enabled and disabled by users. The current implementation polls for break-points being hit in the simulation environment. However, it is desirable that such events occur asynchronously, i.e., when a break-point is hit in the simulation an asynchronous message informs the debugger about the new system state which can be displayed accordingly in the editor window.

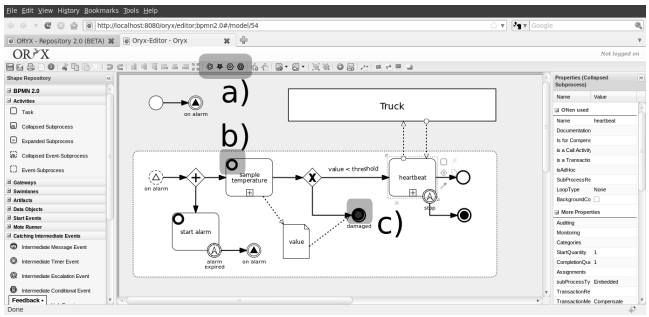


Figure 5.11: Web-based model editor for BPMN modeling, simulation and debugging of wireless sensor networks showing: a) buttons for compile, run, toggle breakpoints, and stepping, b) active break-point, c) break-point hit.

The implementation of the model debugging currently shows the status of a single mote. It is however possible to switch between motes by using their (EUI) identifiers. Breakpoints can be activated for all motes which are described by the same process or only for a single mote. Advanced simulation features, such as dynamically changing the position of a mote, are supported by means of different interfaces, which are not integrated into the model editor. This is achieved for example using the Mote Runner dashboard. An important functionality of the model debugger is that it can attach and query the state of an existing simulation at any time.

5.6.3 Platform API Toolbox

The platform API is integrated as a drop-down menu for tasks and events, as exemplified in Figure 5.12. Users can graphically browse and select the desired API operation for model elements. When the user makes a selection, the corresponding `OperationRef` BPMN property is changed. The input and output parameters as well template assignments are also automatically adapted. Furthermore, if the task involves asynchronous events, they are added as interrupting or non-interrupting border events to the respective task.

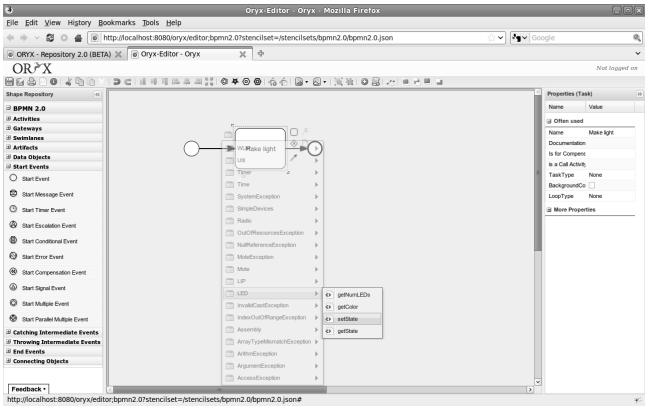


Figure 5.12: For ease of use, the platform API is integrated as a drop-down menu for tasks and events. Users only need to provide corresponding inputs and outputs. Detailed properties can be modified using tabular forms.

The label of the task is initially set to the name of the API operation. Users can change this initial label to more meaningful mnemonics. As described in Section 5.1.3, importing an API is only semi-automatic. To facilitate the import of evolving APIs and different versions of the Mote Runner system, we extended the documentation of asynchronous methods to include the semantic of the different events.

5.6.4 Compiler Hints

With the aid of the symbol table and the debug map, generated by the model compiler, errors and information from all tools are processed and reported back to the modeler directly in the browser window. Warnings and errors are overlaid on top of the graphical model symbols. Crosses are used for marking errors and triangles for marking warnings.

Figure 5.13 shows an example error message which can occur during compilation. In this case, the parameters for a BPMN task did not match the corresponding platform API. The graphical component which is responsible for the error is marked using an overlay in the top-left corner. To maintain an overview of the model, the error or warning text only appears when the user hovers over the respective markings. To help users in solving problems, the messages include hints. The hint message refers to the consistency rules defined in Section 4.2. For example, a hint may indicate which BPMN property is missing or should be filled.

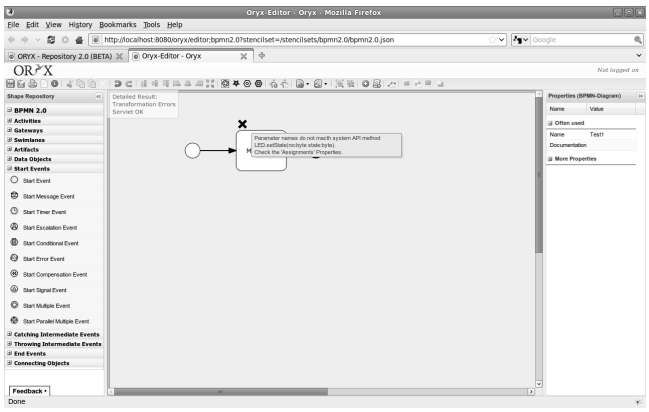


Figure 5.13: The model compilation errors are overlaid on top of the graphical components which caused the error. Messages include hints for solving the model compilation problems.

Summary

This chapter focused on the set of tools required for the development of business-aligned WSN applications. We showed how WSN-aware business modeling tools can be implemented and integrated with a state-of-the-art WSN development environment. The integration produces a coherent set of tools which allows the building of executable WSN applications from models. We will use our implementation of the tools to evaluate the business-aligned methodology proposed by this thesis. A preliminary evaluation of our tools showed that the generated code did not meet the resource requirements of our target platform. Thus, we next analyze possible ways to optimize the resource usage.

Chapter 6

Optimizations

In the preceding chapters, we first analyzed how the behavior of business-aligned WSN applications can be modeled using the BPMN language (cf. Chapter 3). Next, we showed how the models can be automatically transformed to code (cf. Chapter 4) which executes on a particular platform using a corresponding API. We further elaborated on the model-driven approach by describing the required set of integrated tools (cf. Chapter 5) and how the platform API is reflected at the modeling level. However, the transformation from BPMN models to executable code did not consider resource usage in terms of either energy consumption or memory footprint.

Business process models do not and should not perform explicit power management. Low-level power management decisions should be handled efficiently by the run-time platform. However, for best performance, existing WSN runtime systems leave energy management decisions to the applications. These decisions must be implemented manually by developers. Furthermore, to enable the automatically generated executables to compete with manually-coded implementations, the generation step must account for platform-specific optimizations. These optimizations reduce the memory and energy footprint.

Consequently, in this chapter, we first show how the energy management capabilities of the run-time platform can be optimized using a time-based API and an efficient sleep implementation. We refer to these optimizations as *energy micro-management*.[1] Next, we show how the code-generation algorithm should take advantage of the energy micro-management functionality provided by the run-time platform without affecting the BPMN model specification. Furthermore, we describe several compiler optimizations that reduce both the memory footprint and the energy consumption. Although we use the IRIS mote hardware and Mote Runner as examples in this chapter, the concepts presented are applicable to other modern sensor network platforms.

6.1 Resource Usage

Two important resources which directly influence the economic viability of WSN are the memory footprint and the energy consumption of each mote. A smaller memory footprint allows the use of cheaper sensor platforms to implement the

[1] The work related to energy micro-management has been published in [23].

same business process, whereas a lower energy consumption enables a less frequent replacement of batteries for wireless motes, or a reduction on the energy bill for wired motes. In the following, we analyze the costs incurred in terms of memory and energy in detail to identify possible optimization areas.

6.1.1 Energy Consumption

Energy-efficiency is crucial for all types of embedded systems that rely on batteries or energy harvesting as their power source, with wireless sensor networks for long-time monitoring being the prime example. Typical applications are precision farming [10, 105], water management [128], heritage monitoring [25], glacier monitoring [12], or soil vibration monitoring at volcanic sites [178]. Such scenarios describe remote and rough sensing environments. Physical access to the motes and their batteries is often not an option and the sensor nodes have to run on their initial set of batteries or on harvested energy (often not spread evenly in the network and in many cases not satisfying the demand). For a long lifetime, energy must be minutely managed during all phases of operation, including the energy spent while sleeping.

For classical software implementations, the energy expenditure is mainly influenced by the decisions programmers make when developing the applications running on the sensor nodes. At the same time, certain energy is spent by the OS without a direct influence from the developers. For example, a VM-based run-time platform such as Mote Runner will typically have a lower performance due to the overhead of executing byte code instead of native code. Thus, one of the optimization goals in our case is to reduce this overhead; for example, by enabling an efficient and automatic energy micro-management for the run-time platform.

Power Characteristics

To handle power management optimally on behalf of applications, an operating system requires detailed knowledge about the power consumption of primitive operations and individual system components in their various states. In this section, we describe our findings for the IRIS mote hardware. Similar measurements have been done for other mote platforms [44, 140]. In addition to previous measurements, our measurements show the cost of an intermittent wake-up from deep sleep, this is discussed in Section 6.2.1. Also note that the idle (TRX_OFF) radio state is ten times less expensive than using the receiving (RX_ON) state and should be used as default state for the radio as discussed in Section 6.2.2.

First we focus on the current drawn for various state combinations of the radio and micro-controller (MCU) of IRIS motes [106]. The IRIS is equipped with an ATmega1281 MCU [5] running at 8 MHz, with 128 kB Flash memory, 8 kB of RAM, and the RF230 radio-chip [6] for communication. Our measurements with a reference voltage of 3 V are listed in Table 6.1. the transitions between radio states are not instantaneous and in our case typical values are found in the radio-chip specifications [6]. For example, for the radio to transit from the

PRIMITIVE OPERATION	ENERGY COST (μAs)
MCU (only)	
MCU POWER SAVE[2] (deep sleep) for $1s$	9.5
MCU EXTENDED STANDBY for $1s$	280
MCU IDLE w/ ALL peripherals DISABLED for $1s$	2390
MCU IDLE w/ TIMER1[3] ENABLED for $1s$	2460
MCU IDLE w/ TIMER2 ENABLED for $1s$	2510
MCU IDLE w/ ALL peripherals ENABLED for $1s$	3390
MCU ACTIVE w/ ALL peripherals DISABLED for $1s$	5760
MCU ACTIVE w/ TIMER1[3] ENABLED for $1s$	5880
MCU ACTIVE w/ TIMER2 ENABLED for $1s$	5920
MCU ACTIVE w/ ALL peripherals ENABLED for $1s$	6720
Radio (only)	
Radio SLEEP for $1s$	0.02
Radio TRX_OFF (idle) for $1s$	1360
Radio PLL_ON (active) for $1s$	7490
Radio RX_ON (receive) for $1s$	15510
Radio BUSY_TX (transmitting) w/ max power for $1s$	16520
IRIS (system)	
Receiving[4] a 13 byte message (608 μs)	12
Receiving[4] a 127 byte message (4256 μs)	81
Transmitting[5] a 13 byte message (608 μs)	13
Transmitting[5] a 127 byte message (4256 μs)	87
Intermittent wake-up ($\sim 4.3ms$)	6.4

Table 6.1: Average measurements comparing the cost of different operations on the IRIS mote platform with a reference voltage of 3V.

TRX_OFF state to the RX_ONE state, roughly 180 μs are required.

Note that the current drawn for the whole system in deep sleep mode (i. e. when the MCU is in POWER_SAVE mode and the radio is in SLEEP mode) is four orders of magnitude smaller than the current drawn when operating the radio (i. e., either transmitting or receiving a message). Because of such an immense difference in energy consumption, a commonly used strategy for saving energy is to keep the entire system (both radio and MCU) in SLEEP mode for as long as possible, wake up only just before a radio operation is required, and then sleep again as soon as possible. To understand the power consumption of radio operations better, Table 6.1 further shows the average consumed charge when

[2]To wake up the MCU, TIMER2 is clocked from the external 32 kHz crystal. No other peripherals are active in this mode.

[3]When the MCU is ACTIVE or IDLE, TIMER1 is clocked from an 8 MHz oscillator and is typically used for time operations with microsecond granularity.

[4]Radio is typically in RX_ON or BUSY_RX mode and the MCU is in IDLE mode to be able to react to interrupts from the radio and to timestamp received messages with the local system time.

[5]Radio is typically in BUSY_TX mode and the MCU is in IDLE mode to be able to react to interrupts from the radio and to possibly timestamp sent messages with the local system time.

receiving and sending (without acknowledgment) a message of minimum and maximum length (given in bytes), respectively, measured on the IRIS platform.

6.1.2 Memory Consumption

memory types

Apart from energy, memory is another critical resource for WSN. Both volatile (RAM) and persistent (flash or EEPROM) memory are typically scarce for WSN platforms to reduce the cost and the physical size of motes. Volatile memory is used only during execution, when the mote is powered on, and holds the buffers which are required for computations, communication, sampling, and analyzing sensor data. Persistent memory stores the actual application code, the data which is constant or requires infrequent updates, and the information which must be available even after the mote is powered off. Volatile memory is typically further partitioned into stack space and heap space. The stack space is used for keeping the method call-stack frames as well as local temporary variables. For systems with dynamic memory management specific state may be allocated on the heap at run-time. For systems without dynamic memory management, the specific state must be pre-allocated at compile time.

The code generation algorithm presented in Chapter 4 does not consider any optimizations in terms of memory usage. For each communication message and for each sensor reading a separate buffer, placed in volatile memory, will be used. In general, a naive code generation will allocate separate memory space for each operation or resource. As a simplification, we can also assume that each meaningful code statement will require corresponding space in persistent memory. Such a basic approach is inefficient and in some cases may not fit on the target platform.

specific state

For example, if a process model uses three timer events then three timer variables and three callback methods will be generated. The timer variables themselves, the statements which configure and start the timer instances, as well the implementation of the callbacks are all part of the application code. Assuming the timers are used by the rest of the application, the code statements related to these timers will take up corresponding flash space. In general, the software timers available to applications are multiplexed over a single hardware timer. Multiplexing requires a queue which keeps track of the deadline for each timer, as well as the address for the corresponding callback. This information represents the *specific state* particular to each timer instance.

The volatile memory required to keep a specific state is dependent on the particular OS implementation as well as the characteristics of the hardware platform. On the other hand, the memory required for buffers is mainly dependent on the hardware characteristics. For example, a sensor providing a 10-bit ADC value requires two bytes for buffer space.

example memory costs

As an example for memory costs, we consider the IRIS mote running the Mote Runner OS. Similar estimates can be obtained using different hardware and run-time platforms. Table 6.2 shows the memory costs for individual operations and buffers. We analyze both the volatile memory, which is required for execution, as well as the persistent memory which is required for storing the application code. The measurements were performed using the Mote Runner simulation environment.

To estimate the percentage usage, we assume the application has at its disposal half of the total memory resources of the platform. The other half is al-

located by the OS for internal management of various queues, message buffers, as well as scheduling. With this assumption, on the IRIS platform, applications will have at their disposal 4 kB of RAM and 64 kB of flash memory.

RESOURCE / OPERATION	MEMORY COST			
	Volatile (bytes)	% used	Persistent (bytes)	% used
Operation & Variable				
using a single timer	60	1.46	50	0.076
using a single device	30	0.73	24	0.037
using a single radio	65	1.59	33	0.050
Buffers				
maximum radio message	125	3.05	2	0.003
medium radio message	60	1.47	2	0.003
minimum radio message	10	0.24	2	0.003
sensing (10-bit ADC)	2	0.05	2	0.003

Table 6.2: Typical memory usage for individual resources and variables on the IRIS mote platform using the Mote Runner run-time environment. We assume the total volatile (RAM) and non-volatile (flash) memory available for applications is 4 kB and 64 kB respectively.

Based on our measurements, using a single timer costs as much volatile memory as a buffer for a radio message of medium size. The cost represents almost 2% of the available volatile memory resources. Considering the fact that timers are prime elements for BPMN models, the memory consumption quickly explodes. Interestingly enough, performing other operations, such as radio communication or sampling sensor devices, require a similar amount of memory resources.

To sum up, allocating resources for each individual variable, such as a software timer, is expensive in terms of volatile memory and can quickly add up for persistent memory.

Optimization Approaches

Every generated code statement increases the size of persistent memory required by an application. Consequently, less statements are better than more statements. The generated code must have a low flash footprint to fit onto our target WSN platforms. In general, more statements also require more time for execution. These two aspects represent the two well-known, and usually conflicting, optimization goals which most modern compilers support: reduce the size and increase the execution speed of applications. In our case, the size dimension is further split into persistent and volatile memory.

optimization goals speed vs. size

For applications with low-duty cycles speed optimizations have a smaller impact on the energy consumption, because the application spends most of the time sleeping and is only active in short bursts. Conversely, for applications which perform more complicated local processing or filtering, speed optimizations become critical. Thus, for computationally intensive applications energy consumption can be reduced if the computation is performed faster. However, size optimizations benefit both the high-duty and low-duty cycle applications.

Consequently, compact executables, in terms of both flash and RAM, are better suited as they can be loaded and execute on platforms with reduced memory resources.

available optimization strategies

The simplest optimization strategy for the generated source code is to leave the code as is, even if the code uses too many memory resources and is inefficient. In general, compilers are able to perform several optimizations such as eliminate unused variables and dead code, propagate constants, predict conditions in loops, rearrange code statements and in-line functions. In this case, we would rely solely on optimizations performed either by the platform compiler or the run-time.

In addition to the above optimizations, in the embedded VM space, the platform compiler may incorporate a byte-code optimization step [84] which will replace byte-code patterns with more efficient and compact equivalents. This way, the flash memory required by the application is reduced. Another optimization, specific to embedded, VM-based systems, aims to improve the execution time of byte codes using a just-in-time (JIT) compiler [39]. A JIT will compile frequently used byte-codes to efficient native code equivalents. Both classic and embedded optimization approaches are complementary to our work.

semantic optimizations

However, the existing optimization strategies are not able to multiplex buffers or timers, which are dependent on the semantics of the original BPMN model. As a general rule generating less code statements directly reduces the flash space of an application, irrespective of JIT or byte-code optimization steps. Generating "verbose" code and solely relying on JIT or a byte code optimizer simply shifts the problem. An optimizer cannot improve existing code without the knowledge about the system behavior which is available to us at compile time. Consequently, we focus on reducing the flash and RAM usage by generating less source code, based on the semantic information available in the model and the platform-specific APIs.

6.2 Run-Time Power Management

Our model-based approach introduces overhead from two sources. On the one hand, the code generation itself always has an overhead when compared to a manual implementation. On the other hand, some overhead is introduced by using a VM-based run-time. As such, we have to compensate for these effects through optimizations.

In this section, we discuss how to compensate the overhead introduced by the VM through an efficient implementation for the deep sleep phase and an automated energy micro-management for radio operations using a time-based API. Whereas, in Section 6.3, we discuss how the overhead, introduced through the automated code generation, can be reduced by reusing resources based on the semantic information available in the models.

6.2.1 Optimized Sleeping Strategy

Much of the literature seems to neglect that a mote cannot remain in deep sleep mode for extended time periods. Sleep periods are governed by the maximum sleep time span the hardware timer allows. When this timer overflows, the software counter which keeps track of the local system time must be updated.

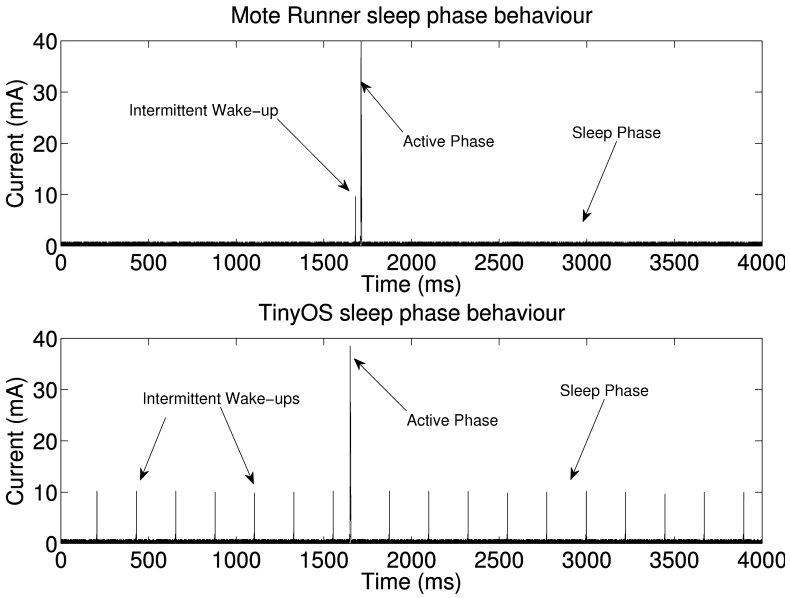


Figure 6.1: A typical deep sleep and communication behavior for Mote Runner (upper signal) and TinyOS (lower signal).

Therefore, the mote must wake up intermittently to update its local time. Typically these intermittent operations include computing the next future deadline when to wake up again, picking a comparator value for the timer module, and finally updating the logical system time. For the ATmega1281 (MCU on the IRIS platform) this has to happen at least every eight seconds. Figure 6.1 shows such an intermittent behavior for an IRIS mote using Mote Runner (upper signal) and TinyOS (lower signal) respectively. The tall spikes represent a period of useful activity, i.e., sending or receiving a radio message. The smaller spikes represent intermittent wake-ups during the sleep phase. A typical intermittent wake-up is shown in Figure 6.2 and consumes half of the energy of transmitting a 13 byte radio message.

energy consumption during sleep

Our measurements show that the energy consumed in a single intermittent wake-up is $6.4\mu C$, which is half the cost of transmitting or receiving a small radio message of 13 bytes (cf. Table 6.1). The energy consumed by an intermittent wake-up is mainly due to the micro-controller wake-up sequence which ensures a stable operation voltage and clock signal. A wake-up sequence starts when an interrupt condition is met and completes when the MCU is active and calling into the interrupt routine. While this sequence is in progress, the current drawn is gradually increased. On the IRIS platform, the wake-up sequence takes on average $4.3\,ms$. During an intermittent wake-up the MCU is in ACTIVE mode for less than 100 μs to update the software system time. Subsequently, the MCU returns to deep sleep.

Figure 6.2 shows a detailed view of a typical intermittent wake-up for the IRIS mote, where the y-axis represents the current drawn in mA and the x-axis represents the time in μs. The sharp peak represents the period during which the micro-controller is active, computing the required values for the next

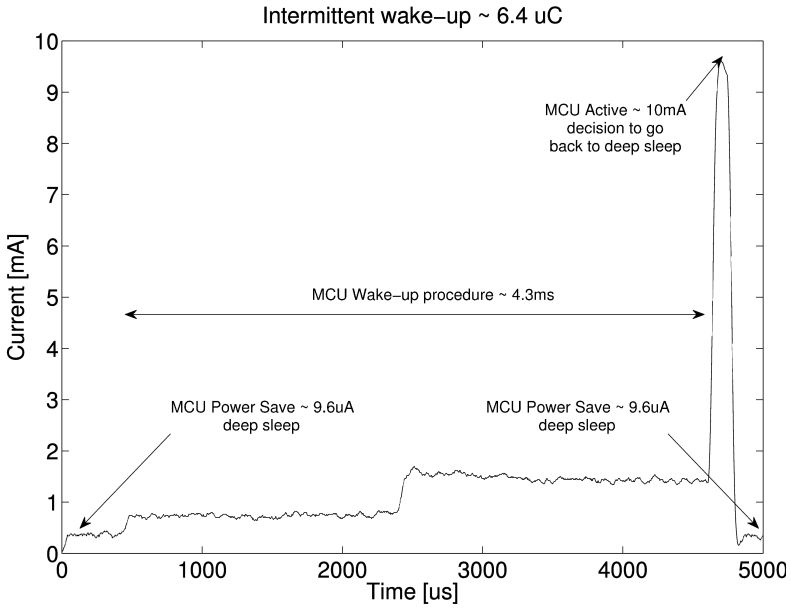


Figure 6.2: A typical intermittent wake-up which occurs periodicaly during the deep sleep phase.

wake-up. The slow rising curve with two levels represents the micro-controller wake-up sequence. The consumed charge is the integral of the curve.

The charge consumed while sleeping $q_{sleep}(t)$ for a time interval t given in seconds is a function of the charge consumed during intermittent wake-ups $q_W(t)$ and the charge consumed in the deep sleep periods $q_S(t)$:

$$\begin{aligned} q_W(t) &= N(t) \cdot Q_W \\ q_S(t) &= (t - N(t) \cdot T_W) \cdot I_S \\ N(t) &= \left\lceil \frac{t}{T_{MAX}} \right\rceil \\ q_{sleep}(t) &= q_S(t) + q_W(t) \end{aligned} \tag{6.1}$$

Q_W denotes the charge consumed by a single intermittent wake-up ($Q_W = 6.4\,\mu C$), T_W denotes the duration of the intermittent wake-up ($T_W = 4.3\,ms$), and I_S denotes the current drawn during sleep ($9.5\,\mu A$). The values for these constants have been measured on the IRIS hardware platform. The number of intermittent wake-ups for a given time period, $N(t)$, depends on the maximum time span for which an implementation may sleep without interruptions, T_{MAX}. This constant is limited by the hardware features which determine the longest sleep interval without imposing a wake-up. These features are: the frequency of the clock used during sleep, the largest timer/counter value, and the largest pre-scaler for the respective timer. A pre-scaler divides the oscillator frequency to allow the timer/counter to reach longer time spans which controls the maximum sleep time. For example, for the ATmega1281, an 8-bit platform with a clock frequency of 32768 Hz (in deep sleep mode) and with the largest pre-scaler value of 1024, the longest sleep interval is approximately $8s$.

energy consumption ratio

The total power consumption decreases if motes wake up less frequently. We denote by η the ratio between energy consumption during wake-ups and the total energy spent in sleep periods. It follows that the theoretical limit of the wake-up to sleep energy ratio η is given by:

$$\eta = \frac{q_W(t)}{q_{sleep}(t)} = \frac{Q_W}{Q_W + I_S \cdot (T_{MAX} - T_W)}. \tag{6.2}$$

Using our measurements, this energy ratio is at least $\eta \geq 7.7\%$, i.e., at least 7.7% of the total sleeping energy is spent waking up and handling the clock. In order to save energy while sleeping, the number of intermittent wake-ups is to be minimized. An optimized sleep strategy reduces the energy consumption and is independent of the application or communication protocol used.

Implementation

The Mote Runner implementation computes a wake-up schedule with dynamic values for the clock pre-scaler and respective comparator value. A precomputed schedule reduces the work of an intermediate wake-up to a single table look-up and immediately going back to sleep. To this end, a knapsack optimization problem minimizing the number of items has to be solved. Items represent wake-ups and they have a finite number of sizes given by the combination of pre-scaler and comparator values. Computing such a schedule introduces a variable delay which must be accounted for when synchronizing with the local system time. To measure this delay we use a hardware timer with a μs granularity.

When switching the MCU between deep sleep and active mode the two respective clocks (given by the external low and internal high frequency) must be synchronized to maintain an accurate local system time. Switching between these clock sources includes clearing the state of the pre-scaler, dealing with possible overflows, and waiting for the two clocks to synchronize. During the synchronization process the OS ensures the MCU is kept in a power-saving state (either POWER SAVE or EXTENDED STANDBY from Table 6.1).

The sleep schedule computations and system time synchronization are time critical operations. The hardware peripherals used for these operations must be shielded from application code. Implementing such a sleep strategy is possible, but challenging, on embedded platforms which allow applications to control hardware resources. In this case, applications striving for energy efficiency must to be omniscient to avoid, for example, disabling a hardware timer which another component requires for sleep operations. Such errors in the sleep implementation may adversely affect time synchronization. We implemented the optimized sleep strategy for the Mote Runner OS/VM because it provides encapsulation by design.

Our discussion used the IRIS mote as an example. However, the described sleep optimizations can be implemented on other modern hardware platforms. A necessary requirement is a dynamic and deterministic pre-scaler to control the external low-power clock.

6.2.2 Micro-Managing Communication

In this section, we introduce a platform-independent abstraction for the radio API which enables the OS to minutely manage the power state of the radio chip

on behalf of the application. Developers implementing communication protocols can focus on protocol functionality while energy management decisions and respective transitions are automatically handled by the radio abstraction in the OS, even for very short time intervals. The main goal of the radio abstraction is to simplify development effort and improve energy efficiency.

design considerations

As a main design criteria our abstraction generally includes *time* values to schedule critical, externally observable actions such as radio transmissions. This empowers the run-time to manage resources in the most energy-efficient way on the given hardware. Developers no longer need to be concerned with manual power management. The key difference is that developers simply state the exact time when some action should be performed, without knowing or guessing hardware-specific transitions. It is the run-time that ensures devices such as the radio are operational and in the desired state when the application requires them. By default, the micro-controller, radio, and all sensor devices are kept in the most energy-efficient power mode with the decision of when to power on and off certain devices being made on a microsecond basis depending on the API calls from the user application. Transitions between radio states are not instantaneous, such delays are platform-specific and can be configured based on hardware measurements. For example on the IRIS platform it takes $\sim$880 μs for the radio to transit from the SLEEP to the TRX_OFF state. This configuration is part of the OS implementation and not the application code.

Our radio abstraction consists of the following elements: a) internal operating states which may map to different platform-specific radio-chip states, b) method calls which trigger some action which changes the internal state, c) callbacks which notify application code about completed events, and d) internal state changes due to passing of time. The radio abstraction ensures that actions are performed, callbacks are invoked and the internal state is changed in time to meet the deadlines requested by the application code. Normally, an application starts a task (e.g., transmit) by calling an API method. The task execution takes some time and upon completion an application-specific callback will be invoked by the OS.

Figure 6.3 shows all possible radio operation states and respective transitions. The states are abstract but can be mapped to different modern radio-chips such as the RF23x or CC24xx. In this discussion we use the RF230 radio chip as an example. Figure 6.3 a) represents the main radio state diagram used for micro-managing the radio power consumption based on the application requests using time information. The sub-diagrams T and E shown in Figure 6.3 b)-c) are embedded in the main diagram (a) as squares. Figure 6.3 d) shows additional radio state transitions. Generic radio states managed by the OS are shown using ellipses (stable states) and circles (volatile states). API function calls are depicted using rectangles while callbacks to application code are shown with rhombuses. API methods take precise points in time as parameters while callbacks report the exact completion time and status.

logical radio states
stable vs. volatile

We distinguish between *stable* and *volatile* operating states. Individual methods of the radio API may be called only if the radio is in a stable state. Calling methods while the radio is in a volatile state, where this operation is not permitted, will result in an exception being thrown.

The stable states are shown using ellipses: SLEEP (sleeping), RXON (background reception of messages), RXOFF (reception period ended), and ACTIVE (ready to quickly transmit or receive). The volatile states are shown using cir-

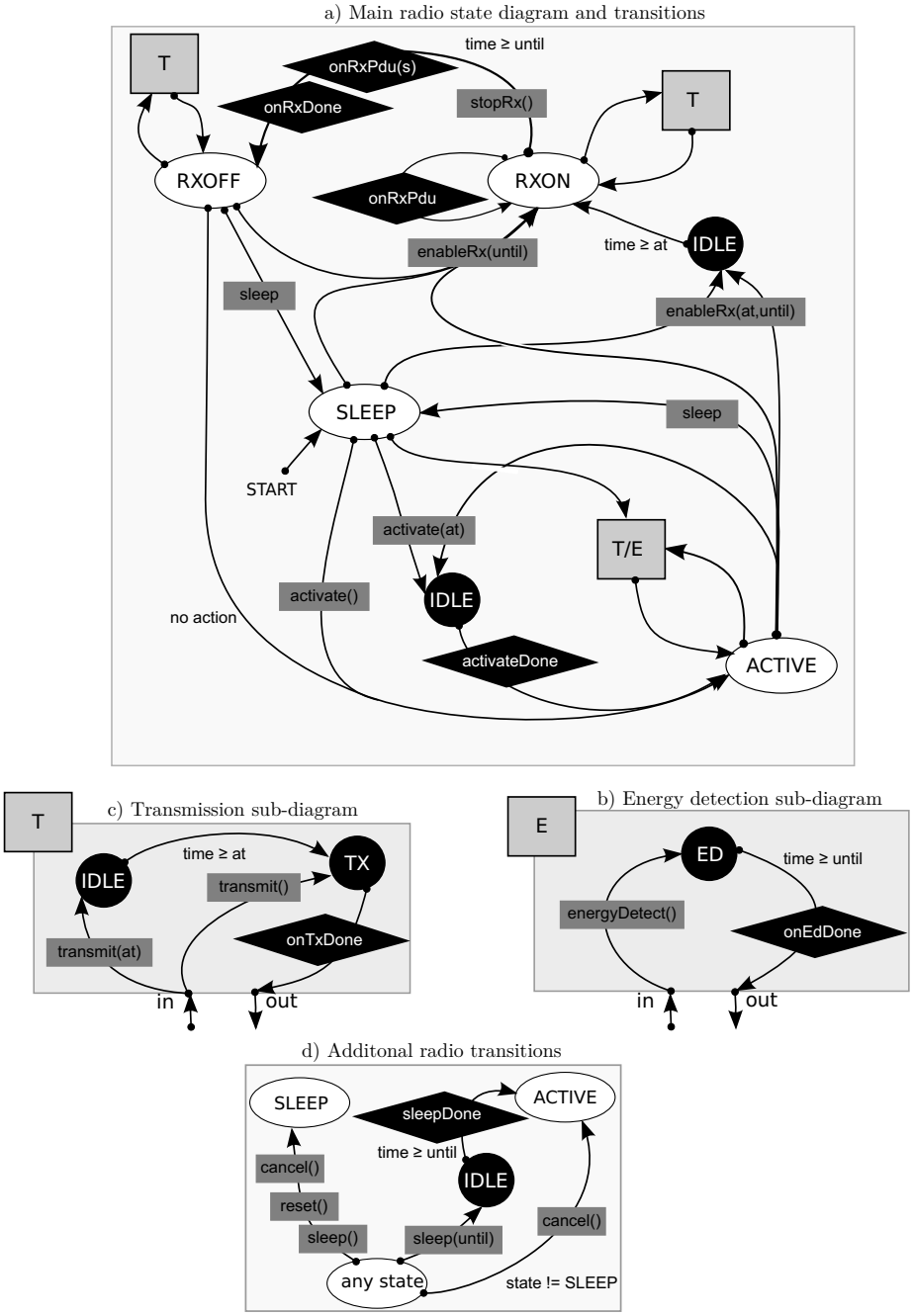


Figure 6.3: Radio state diagrams for communication micro-management. (b)-(c) Sub-diagrams T and E are embedded in the main diagram (a) as squares. (d) Additional radio state transitions. Generic radio states managed by the OS are shown using ellipses (stable states) and circles (volatile states). API function calls are depicted using rectangles while callbacks to application code are shown with rhombuses. API methods take precise points in time as parameters while callbacks report the exact completion time and status.

cles: TX (transmit), ED (energy detection), and IDLE (preparing the execution of an action). The T and E sub-diagrams are depicted using squares and show the transitions performed when the respective transmit and energy detection actions are requested. The sub-diagrams blocks are expanded in Figure 6.3(b)-(c) and are collapsed and embedded in the main radio state diagram in Figure 6.3(a) to simplify the graphical representation.

In addition, from any state, all operations may be canceled and the radio may be instructed to go to sleep. A sleep request may take a point in time as a parameter which represents the time until which the radio should be kept asleep. In this particular case, the OS ensures that at the specified time the radio is in the ACTIVE state and the callback `sleepDone` is invoked, cf. Figure 6.3(d).

To save energy, the initial operation state for the radio is SLEEP (directly mapped to the same state from Table 6.1). From this state, the application may request one of the following actions: `activate` the radio, `transmit` a message, enable the receiver (`enableRx`), or perform an energy measurement on the radio channel (`energyDetect`).

time-based application requests

Applications request certain actions using API function calls which are depicted using rectangles in Figure 6.3. API calls take a point in time as a parameter which specifies the precise time at which the action should be executed. Without a time parameter the start of the action is triggered immediately. In the case of a transmission, this parameter represents the exact time when the transmission of the radio frame must start. Further energy considerations allow the system to select the power level for each transmission individually. To simplify application development a transmission mode can be specified such as a transmission with a single clear channel analysis (CCA), or a back-off mechanism such as carrier sense multiple access (CSMA) with corresponding parameters. The OS will take care of starting any necessary wake-up procedures sufficiently ahead of time so that the desired target state is entered and the requested action is started at the specified point in time. Such an abstraction enables the same application code to execute on heterogeneous platforms. Moreover, a simulation environment can use the same timings for radio transitions to match the behavior of the corresponding hardware platform.

The generic, volatile IDLE state is entered whenever a scheduled action with a specified execution time is ahead. This state may map to any of the actual RF230 radio-chip power states (SLEEP, TRX_OFF, or PLL_ON from Table 6.1) depending on the time remaining until the time of execution, the current state of the radio hardware, the desired hardware state, and the hardware-specific transition times. For energy considerations, on our example platform, the ACTIVE state corresponds to the TRX_OFF state in Table 6.1.

callback notifications with timestamps

Applications are notified of completed actions using callbacks which are depicted in Figure 6.3 using rhombuses. Before the callback is invoked (i.e., control is passed to application code) the radio transits to the next state, according to the state diagram. In other words, when a callback is invoked the radio operation state is indicated by the arrow head in Figure 6.3. For example, the radio is already in the ACTIVE state when the `activateDone` callback is invoked. During the callback the application code may request a radio action which will trigger the respective internal transitions. If nothing is requested the OS leaves the radio in the same operation state, because it cannot predict what the application will do next. After the callback is invoked the radio operation state

might change depending on the actions specified by the application code.

For applications to appropriately recover from failures, they must be informed if certain requests failed and what the exact reason for the failure was. Thus, callbacks are always executed and report the completion status of the requested action. The completion status may have several values depending on the requested action. For example, in case of a transmission several errors could happen: no channel access, the time specified for the transmission could not be met, no acknowledgment was received, etc. An exception to this rule are the `cancel` and `sleep` calls without a time parameter: they reset the state of the radio and remove any pending callbacks.

To simplify application scheduling and state maintenance, callbacks report the exact *timestamp* for the completion of the task or the occurrence of the respective event.

further considerations

To save energy, energy spent while receiving messages should be minimized. Thus, the radio API allows the programmer to specify a precise start time when to reach the RXON state and an end time when to exit the RXON state using the `enableRx(at,until)` method.

The `TX` and `ED` states represent volatile radio operation states, where the radio chip is transmitting respectively measuring energy on the wireless channel. When the respective callback is invoked the radio returns to the originating state. For example if transmitting a message from the RXON state, after the transmission is finished and before the callback is invoked the radio is set back to the RXON state. If the application decides to transmit a message from the RXOFF state the radio is put back to the RXOFF state after the transmission is complete.

To complete the API further functions are required such as setting the radio channel used for communication, querying and setting radio address information used for filtering. The full description of these functions and complete API is available in the Mote Runner documentation. As a general rule these calls are synchronous and an application may invoke them only in a stable state.

The radio abstraction described in this section is fully implemented in the Mote Runner platform. Yet, a similar implementation is possible for other sensor network operating systems such as TinyOS [165] and Contiki [37]. We will show that this API enables a VM to perform a TDMA protocol as efficient as a custom OS in terms of energy expenditure.

6.3 Generated Code Optimizations

Having an optimized execution platform, which is able to automatically micromanage energy on behalf of the application, is the first step towards an efficient execution of business processes on embedded WSN platforms. However, to benefit from such features, the generated code must use the specific time-based API. In addition, further optimizations should reduce the footprint of the generated application for both persistent and volatile memory by reusing resources such as buffers and timers. These optimizations are complementary to the previous considerations for the underlying run-time.

6.3.1 Reusing Buffers

In general, communication requires space in volatile memory for buffers which encapsulate the data to be transmitted. The pattern-based compilation algorithm described in Section 4.3 generates for each sent message a separate buffer. Such an approach quickly explodes with the number of transmitted messages. However, a typical sensor node only has one wireless radio device which can only send one message at a time. Consequently, a single buffer may accommodate multiple transmissions. The size of this transmission buffer is simply the size of the maximum sent message which is given by the sum of the sizes of each encapsulated data item, based on the data types.

multiplexed transmission buffer

We propose using a *single buffer* for all transmissions. This reduces the transient heap usage and, at first sight, also the flash space occupied by the application. The flash space is reduced because the code requires only a single variable which holds a pointer to the buffer instead of multiple variables. However, before transmission the buffer must be updated with both the header and the payload information. The header specifies the destination of the message and transmission method, e.g., broadcast, acknowledged. The payload includes the serialized data bytes to be transmitted. The code which performs the update again increases the flash size. Depending on the complexity of the header and payload update the increase in the code size may dwarf the initial flash reduction (due to less variables). Moreover, to improve execution time, updating the message buffer should only be performed if required. In other words, if the transmission sends the same content, no update should be performed.

Thus, a trade-off exists between using several buffers or a single buffer. Multiple buffers improve execution efficiency at a lower flash cost, whereas a single buffer adds a penalty on both execution and flash size. However, the single buffer approach reduces RAM usage to a minimum. For the platforms we target, RAM usage is more critical than flash space. For example, on the IRIS mote, there is 16 times more flash available than RAM. Consequently, the single buffer approach is appropriate.

multiple, OS-managed reception buffers

Theoretically, reception buffers can be treated in the same way as transmission buffers. However, in general, WSN run-time platforms provide the application with several OS-managed buffers for reception. The rationale is that several messages can be received even before the application has a chance of processing them. Thus, the application does not allocate any buffers for reception, which is also reflected in the generated code (cf. Figure 4.4). Instead, a pointer to the message is passed as a parameter to the reception callback. However, the application may wish to save the received data in some temporary buffer. In this case, the temporary buffer will not be optimized according to the single buffer scheme described previously.

multiplexed device buffers

By contrast, the single-buffer principle can be applied to sensor and actuator buffers which the application needs to provide (and allocate) for data readings. To determine whether certain sensing or actuating events or actions can occur potentially in parallel, and thus require separate buffers, we can use the parallelism detection algorithm presented in Section 4.5. In this case, instead of sub-processes, the algorithm searches for parallel execution of the same platform API which corresponds to tasks or events for communication, sensor readings and actuators. If the tasks are executed serially the corresponding buffers can be shared. Otherwise, the corresponding buffers cannot be shared.

6.3.2 Reducing The Number of Timers

Timers are a common feature provided by the underlying OS to control various periodic and one-shot alarms. They require keeping some specific state which uses volatile memory. As described previously, using less timers in the generated code reduces both the volatile memory as well as the flash space occupied by the application.

As an optimization step for the code generation, we propose replacing costly timer variables with time-based parameters, where appropriate. This optimizations reduces the number of timers and can be achieved by identifying patterns in the BPMN models which use timer events. The timer events in the model specify precise moments in time. For example, when the following activity should be performed.

The code generation introduced in Section 4.3.4 does not consider such patterns. Instead, for each BPMN timer event, a timer variable is generated (cf. Figure 4.13), which needs to be configured, next the OS timer is started. Upon the expiration of the alarm the corresponding callback is invoked which finally calls the actual action to be performed.

A common pattern for communication is, for example, a timer event followed by either a transmission event, as shown in Figure 6.4(a), or a reception event, as shown in Figure 6.4(b). The above pattern remains valid even if the send and receive events are replaced by corresponding send or receive activities. Moreover, the same principle can be applied to any activity whose underlying API method accepts a time value as a parameter.

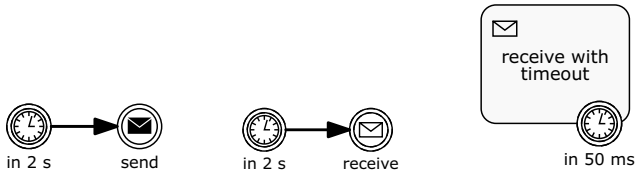
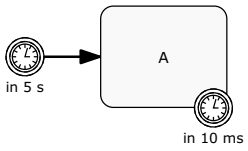


Figure 6.4: Common timer patterns for communication which are translated to single calls using parameters in the time-based radio API

Timeouts are another frequently occurring pattern. They are described using timer events attached to the border of activities, as exemplified in Figure 6.4(c). Timeouts can also be expressed using event-based gateways followed by corresponding timer events as shown by the examples in Figure 3.13. Such timeouts can again be specified directly using the previously described time-based radio API or any API which supports a timeout parameter.

By recognizing timer patterns at the model level, the moments in time when actions should be performed, e.g., radio reception or transmission, are now specified directly using the corresponding parameters provided by the time-based API. The generated code is thus free of costly OS timer variables and corresponding callbacks.

Figure 6.5 shows an example model where a timer is used to start an activity




```
1  void (*A_continue)();
   void (*A_timeout)();
3
   process (){
5    A_continue = &after_A_done;
     A_timeout  = &when_A_timeout;
7    long start_time = CURRENT_TIME + SECONDS(5);
     long end_time = start_time + MILLISECS(10);
9    A_start(start_time, end_time);
   }
11
   void after_A_done(){
13   // invoked if activity completed normally
   }
15
   void after_A_done(){
17   // invoked when activity timeouts
   }
```

Figure 6.5: Code generation example which directly invokes a timed-based platform API method `A` using a `start_time` and an `end_time` parameter instead of generating costly variables and code for OS timers. Thus, considerably less code is generated when compared to the pattern shown in Figure 4.13.

in 5 seconds from now. Once the activity is started a timeout for its completion is specified to abruptly interrupt the activity in 10 milliseconds. The generated code first deals with the setup required for asynchronous operations (lines 1-6). Next, the `start_time` is computed (line 7), using the current time, and is directly passed as a parameter. The `end_time` is computed using the `start_time` as reference to which the timeout specification is added (line 8).

Note that reducing the number of timers as described above is a platform-specific optimization which is only possible for the API methods which provide corresponding *start time*, *end time*, or *timeout* parameters. Also note that the effect of the optimization is proportional to the number of activities which require precise moments in time. Such activities are frequent for example in TDMA-based sensing or radio communication.

In certain cases, OS timers can be reduced even further. For example, if two timer events are not used in parallel, according to the algorithm in Section 4.5, they can share the same OS timer.

6.3.3 Preferring Static Execution

In general, if a model can be compiled using the static code generation, because there are no twin tasks executing in parallel, this method should be employed instead of the dynamic mode. The static compilation method reduces the stack and heap requirements and requires no additional flash space for the BPMN run-time.

As a rule, the algorithm presented in Section 4.5 can be used to detect twin tasks executing in parallel. In such cases, the algorithm will warn users. For a general solution, the default behavior for the compiler is to use the dynamic generation mode. However, if the modeler knows that in practice parallelism cannot occur, he can force static compilation. This is achieved by using the corresponding compiler flag (cf. Section 5.2).

6.3.4 Removing Dead Code

The compilation algorithm can generate code independently for different parts of the model. This code includes not only variables but also the functions (e.g., stop and pending) which implement the execution semantics of BPMN models (cf. Section 4.2.2). However, not all generated code is necessarily used for the execution of the application. Unused code may involve both methods and variables.

A typical example is the `stop` method for a task. Assume, for a given model, the method is never invoked because there is no corresponding interrupting event. The same is true for variables which have been generated for data items which are specified in the model but are never used by any tasks or conditions at gateways. Thus, in a final compilation step, functions and variables which are not used can be removed entirely.

To remove unused code, we rely on the warnings provided by the platform compiler. The warnings are analyzed and the corresponding code is completely removed from the generated source file before a second compilation step. The dead code elimination has only a marginal effect on the volatile memory usage. However, dead-code elimination can significantly reduce the size of the flash space occupied by the application.

Dead-code elimination is in theory possible for both the static and the dynamic compilation mode. However, this optimization pays off more for the static case, because the number of generated functions is reduced to the minimal set which still correctly implements the process execution according to the model specification. By contrast, the dynamic case shares the implementation of the BPMN run-time which implements the BPMN functionality such as canceling sub-processes. Even if not called, a method remains part of the BPMN run-time which is implemented as a separate assembly.

6.3.5 Further Considerations

To reduce the flash size, further optimizations are possible. For example, if a gateway does not require splitting or merging, the corresponding method need not be invoked and thus not generated at all. Splitting or merging may not be required, for example, if there is exactly one outgoing and one incoming sequence flow. Instead, the compiler can simply generate the next statement from the abstract syntax tree.

null callbacks

Other sources for optimization are events, e.g., attached to the border of activities, that do not continue with an outgoing sequence flow. In this case, the (empty) method corresponding to the event is not generated. As such, null delegates are passed as the callback parameters for the API methods which may trigger the corresponding event. The same principle can be applied to all asynchronous operations which require a callback but for which the model specifies no continuation using sequence flows.

lazy instantiations

To reduce the maximum heap usage, the generated code performs lazy instantiation. Thus, new objects are created only shortly before they are needed. This applies to common objects used for OS timers and devices as well as new instances for the classes implementing sub-processes. However, the instantiation and initialization of new objects increase the execution time.

The other extreme is to initialize all required objects and buffers at the start

of the application, e.g., in the constructor. In this case, pre-allocated resources such as timers and buffers are never discarded, because there is always a static reference to them. Such an approach is simpler to implement and has a reduced execution time compared with a lazy instantiation and initialization approach. However, the RAM space may simply not suffice for such an approach. Consequently, our implementation uses a hybrid approach in which lazy instantiation is used for objects (e.g., sub-processes, devices) and space is pre-allocated for the single transmission buffer.

In general, a software developer has better knowledge of when resources are used, and more importantly, when they are not needed. Thus, manual control can outperform the automatically generated code which uses a lazy instantiation policy.

Summary

In this chapter, we analyzed the energy and memory expenditure and proposed several optimizations to reduce these costs. The optimizations address the underlying WSN run-time platform as well as the code generation. Both optimizations are necessary for meeting the requirements imposed by the target platform. Models can thus focus on describing the desired business functionality rather than low-level resource management details. We can now concentrate on the evaluation of our methodology.

Chapter 7

Evaluation

In this chapter, we evaluate our proposed model-driven methodology with respect to the business requirements set forth by this thesis, namely, alignment, flexibility, confidence, and efficiency. For the evaluation, we use our own implementation of the integrated modeling tools described in the preceding chapters. Note that the first three requirements are qualitative, whereas the last requirement is quantitative.[1] Moreover, the qualitative requirements can be summarized as to whether our modeling approach provides sufficient generality and *expressiveness* to address the issues of typical WSN applications. Before we discuss expressiveness and efficiency in the next sections, we briefly summarize how the qualitative requirements are addressed by our methodology.

- *Alignment* is provided by our methodology through the integration of WSN-specific models directly into the business modeling tools (cf. Chapter 5). Thus, the behavior of WSN applications is now visible in business process models from the design phase on. Furthermore, the details of WSN applications can be controlled (cf. Section 3.2), discussed, and understood by domain specialists with a background training in a process modeling language such as BPMN.

- We addressed the dual aspects of *flexibility*, namely, changes in the business requirements and changes on the WSN side. If the business requirements change, a new process model is required to reflect the changes. Afterwards, the corresponding WSN application can be automatically generated (cf. Chapter 4) using the integrated compiler tool (cf. Section 5.2). Furthermore, the new application can be pushed into the network (cf. Section 5.4.5) using over-the-air programming. Conversely, if the underlying WSN platform changes, due to improvements in hardware or better communication libraries, the new platform API can easily be imported (cf. Section 5.1.3) into the model editor.

- The *confidence* aspect is addressed by allowing the process models to be tested and debugged directly from the model editor (cf. Section 5.3) using an accurate simulation environment (cf. Section 5.4.2). As we use the same application binary which also executes on hardware motes, the

[1] Some of the results have been partially published in [21, 23, 24] or submitted for publication. The contents of this chapter represents a more thorough analysis.

confidence in the correct behavior of the business process, with respect to the WSN part, is increased. In this case, we need to assess whether the same behavior which is observed in the simulation is matched by hardware deployments. For this purpose, we employed testbeds using off-the-shelf IRIS, RAVEN, and RZUSBSTICK mote hardware.

7.1 Expressiveness

Even though expressiveness is widely used in the literature as a distinguishing feature for many newly introduced programming languages [42, 69], there is no widely accepted formal definition [104]. Formal approaches, such as [104, 112], compare the expressiveness of two programming languages using a translation which maps programming constructs from one language to the other. If a mutual translation exists, the two languages are equally expressive. In essence, our pattern-based compilation algorithm describes a translation in one direction, from BPMN to an imperative programming language, e.g., Java. However, the reverse transformation, from the full Java language to BPMN, is not possible for all Java constructs, e.g., inheritance, templates, or variable-length arguments. Thus, BPMN is not equivalent with Java in all programming constructs.

Turing complete

In the most general terms, expressiveness refers to the notion of Turing completeness. This criterion determines whether a system that manipulates data based on rules can simulate a Turing machine. Most modern imperative programming languages, such as C, Java, or BPEL [60, 168] are Turing complete. In this thesis, we used BPMN, including the non-graphical properties, as a form of graphical programming. As BPMN provides imperative constructs which one also encounters in modern programming languages, such as for loops, if-then-else structures, as well as data variables, it can be shown that BPMN is a Turing complete language. Thus BPMN can be used to describe general computational algorithms. However, BPMN is not a "Turing tar-pit in which everything is possible but nothing of interest is easy"[138]. In this thesis, we have shown how BPMN can be used to describe meaningful WSN applications.

In addition to the above computability requirements, BPMN provides constructs for parallelism and synchronization, which are features typically found in multi-threaded programming environments such as Cilk [46]. Parallel behavior thus becomes visible in models, which are easier to deal with than event-based code [11]. Moreover, dealing with events and communication is greatly simplified in comparison to manually-coded applications which require preparation code for setting up buffers and adjusting the required parameters. In our methodology, the communication parameters are automatically generated, which enables models to focus on the core functionality rather than the technical infrastructure. Nevertheless, parameters for communication can still be minutely controlled using the non-graphical properties.

practical aspects

In practical terms, the expressiveness of a (programming) language is intuitively related to the desired abstraction level. In other words, a language provides sufficient expressiveness if the program logic can be represented using the same concepts as in the application domain. For example, if mathematical functions represent the desired abstraction, assembler language does not provide the right expressiveness as it requires extra preparation code which hides the

actual logic of the functions.[2] In this case, functional languages, such as Haskell or LISP, would be optimal candidates because they allow the logic to be expressed using the same concepts, namely functions. Conversely, changing a bit in an MCU register is not expressible with a declarative programming language such as Prolog. In this case, assembler or C would be the optimal choices.

By its design, BPMN clearly meets the abstraction level for the application domain of business processes. Similarly to declarative programming languages, controlling individual hardware-specific parameters is not expressible using our BPMN-based methodology. This limitation is a design choice which allows our solution to be portable across different hardware platforms (cf. Section 3.2.2).

compactness of representation

Often expressiveness is also associated with the compactness of the representation. In our case, one measure that indicates compactness is the ratio of the number of symbols required by BPMN models to the lines of source code required for implementing the same functionality. Using example models for WSN applications, we found that the average ratio of graphical symbols to lines of code is roughly one to five. In other words, for each graphical symbol, on average five lines of code would be required. For this analysis, we considered all graphical symbols, i.e., not only pools, tasks, events, and data but also the sequence flows, communication flows, and data flows.

7.1.1 Extension Equivalence

Since its standardization, several extensions to the graphical symbols (or stencils) of BPMN have been proposed. The extensions address both the WSN space as well as other domains, e.g., processes based on RESTful [137] interfaces or support for social network [14] business activities. Related to our work on BPMN in the WSN space, two sets of extensions have been proposed [110, 159] to better express the relationship between processes and the physical world. However, in this thesis, we argue that for describing the behavior of WSN applications, and in particular for generating executable code, no extensions to BPMN are required.

physical annotations

The extensions in the first set define annotations which describe physical properties of tasks. Such properties include the potential failure and the certainty of the provided information [110]. However, potential failure can be expressed in a fully BPMN-compliant manner using error events, which is also our approach (cf. Section 3.2.1). Moreover, it is not clear whether the certainty of information is a static property or whether it is computed at run-time. In the first case, it can be included as a constant, non-graphical property of BPMN tasks, instead of graphical annotations. In the second case, certainty is simply a data value which is computed during execution. This information can be added to the message payload and sent to other entities. Such an approach is again compliant with the BPMN specification and is supported by our methodology (cf. Section 3.2.3).

In line with the above annotations, [159] proposes two new BPMN task

[2] Using assembler code, multiple steps are required for implementing a function that simply adds two variables and writes the result to another variable. Because variables are stored in memory, prior to adding two numbers, the values from the two corresponding memory locations must be placed in registers. Afterwards, the addition, which represents the main logic, is performed using the registers. Subsequently, the result must be written to the memory location.

types which distinguish between a sense and an actuate behavior. However, such extensions do not increase the expressiveness in terms of the execution behavior of WSN applications. In our approach, we treat sensing and actuating in the same way for both modeling (cf. Section 3.2.1) and code generation (cf. Section 4.3.3). In our case, the distinction between sensing and actuating is abstracted by the platform API (cf. Section 3.2.2).

physical entities

The second set of extensions addresses the physical "entity of interest" (cf. Section 3.2.3) which is represented using a new stencil [110]. The new stencil can be associated with tasks to show that they act on or sense properties of the corresponding physical entity. Following the same line of thought, [159] introduces a similar stencil to visually distinguish physical entities or objects. However, BPMN already provides a standard property that specifies whether a data object is purely informational or physical. Different types of data objects, including persistent data (cf. Section 4.3.1), are fully supported by our approach. Furthermore, if the physical entity of interest is capable of communication, it represents a standard BPMN pool (or entity) and should be modeled accordingly.

In summary, the same behavior is expressible using our proposed methodology, without extending the graphical stencils provided by the BPMN language. The proposed extensions mainly highlight physical aspects in a visual form. Standard BPMN symbols suffice for specifying the behavior relevant for execution.

7.1.2 Application Domain

To show the generality and expressiveness of our BPMN-based methodology for our application domain, we used our integrated tools to describe several dozen models corresponding to typical WSN applications. We use the classification of WSN applications defined in [8] which considers the following characteristics: mobility, sampling, transmission, actuation, interactivity, data interpretation, data aggregation, and homogeneity. Based on existing applications, these characteristics were grouped to form so-called *archetypes* with specific characteristics.

data-gathering archetype

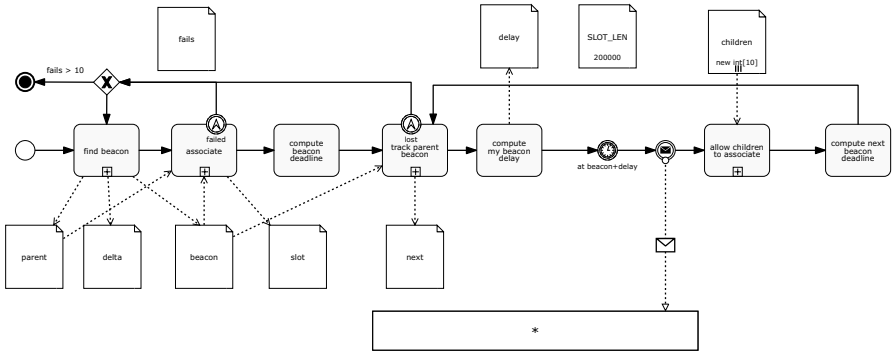
The first archetype is the most common, i.e., it matches the largest set of existing WSN applications, covering more than 30% of the applications [8]. In this case, nodes are stationary, sensors are sampled periodically, and the data is transmitted periodically. Furthermore, there is no actuation nor interaction with the sensor nodes. However, there might be some data interpretation as well as aggregation in the network itself. Moreover, the networks are homogeneous in that all nodes in the network run on the same hardware and execute the same application. This archetype is concerned foremost with "data gathering". Different abstractions such as the SwissQM [117] have been proposed to abstract from the WSN and concentrate on the functionality of the application.

To show that we can integrate existing libraries and different APIs into the modeling tools, we modeled *Gather*, which is a high-level data collection application using a decentralized multi-hop library for the communication. The library is written in Java and is part of the Mote Runner distribution. The API provided by the library was imported into the modeling toolbox.

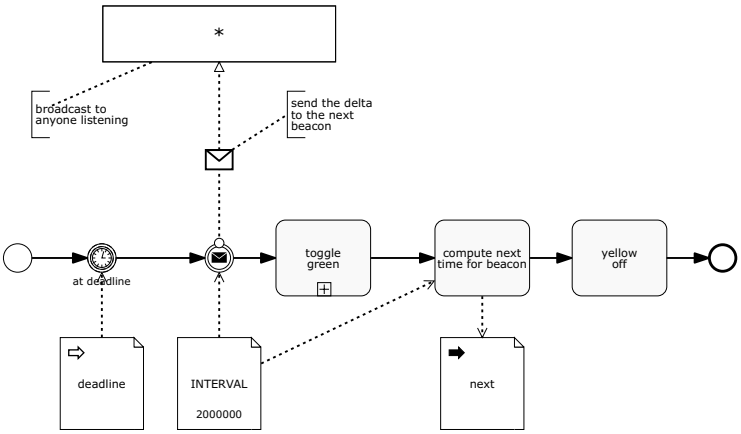
monitoring archetype

The second archetype covers more than 25% of the existing WSN applications and shows similar characteristics [8], but with the following differences.

For this group of applications, sampling may be a hybrid approach, i.e., both periodic and event-driven. Whereas data transmission is always event-driven, and the data is always interpreted in the WSN. This archetype can be summarized using the "monitoring" keyword. A representative application is the parcel-monitoring scenario described in Section 3.1, which we refined only using models. Particular characteristics of this application are a low duty cycle and a fixed topology; the parcels are always contained within a mobile truck. Thus, our networks configuration uses a single-hop star topology with a *Truck* and several *Parcels*. This application is similar to the scenario described in [110] which monitors orchids and adapts the selling prices according to the freshness of the merchandise.



(a)



(b)

Figure 7.1: (a) The top-level process describing the behavior of a `Slave` node in a TDMA protocol which first finds a parent node, then continuously tracks beacons from the parent. (b) A reusable sub-process for broadcasting a beacon message at a specified deadline; LEDs are used to mark externally visible behavior.

To visually control and monitor the parameters of the parcel-monitoring

application, we built a Web-based interface based on Comote (cf. Section 5.4.4). The entire application currently is a permanent, long-running demonstrator in the Industry Solutions Lab (ISL) at the IBM Research - Zürich Laboratory (ZRL). The parcel-monitoring application is also part of the examples in the standard Mote Runner distribution.

low-level communication and synchronization

In addition to the different archetypes, we modeled different communication protocols including TDMA and CSMA approaches. This shows the general applicability of our modeling approach, which allows even the expression of low-level communication and synchronization details. As an example, we modeled a multi-hop TDMA-based protocol where the *Master* root sends beacons which are re-broadcast in assigned slots by *Slave* nodes. The protocol builds a tree structure in which each node can support a fixed number of children.

Figure 7.1 shows two models which deal with the network formation and communication aspects of the *Master* and *Slave* communication protocol. Figure 7.1(a) describes the high-level behavior of a *Slave* node which participates in a beacon-based TDMA network. The network protocol allows nodes to associate dynamically. If the association fails, the node will repeat the attempt for a fixed number of times. Upon successful association, a node will start tracking the beacons from its parent node and send its own beacons. The beacons are sent in predefined slots of 10 ms, to allow other nodes to associate to the network. If a node does not receive the beacon message, it tries to associate again to the network by actively searching for beacons from a parent in the vicinity. Figure 7.1(b) shows a reusable sub-process for sending beacons at specific deadlines. The same sub-process is used by both the *Master* root node and the *Slave* nodes. Further refinements of these models are included in the Appendix. The protocol was modeled entirely, i.e., no library was used to implement the functionality. This can be a rather tedious task, especially when dealing with computation-intensive operations and searching elements in data structures.

Lastly, to clearly expose the systematic overhead of our approach, we also modeled several base cases. A typical example is *Blink* which toggles an LED every 500 ms.

To verify the behavior of the generated applications we used predefined network configurations and mobility models in the Mote Runner simulation environment. For example, in the case of the TDMA protocol described above, motes moved into and out of their respective radio range. In this way, different execution paths, such as the re-association in the above model, were triggered.

7.1.3 A Note on Modeling

An interesting aspect of BPMN models which describe WSN applications is that there is no end event on the top-most process level. This is because a WSN application is a long-running, continuous process. For our set of WSN applications, we found no scenario in which the compensation event was required. Moreover, our models discourage the use of the BPMN signal event type as this represents an out-of-band communication mechanism which bypasses wireless links. The signal event propagates both across sub-process hierarchies and pool boundaries. Instead, we prefer the escalation event, which can only be caught once, and propagates upwards in the process hierarchy.

With respect to the different gateways provided by BPMN, we used all types[3] for control flow and synchronization. In particular the event-based, exclusive, and parallel gateways are used frequently, whereas the OR gateway is used rarely. In general, our models use parallel gateways to allow synchronous activities to execute while asynchronous operations are being started. For our application scenarios, the number of gateways decreases for the process models at higher abstraction levels. The process models at the top-most level tend to be a linear chain of tasks and sub-processes.

Because of the reactive nature of our models, we make extensive use of the different types of events provided by BPMN. In particular the escalation, timer and message events are among the most frequent components in our models. In addition, we used the error event for critical failures.

For execution we require maintaining state in our models. Hence, we employ a considerable number of data items for volatile variables and data stores for persistent variables. For passing data inputs and output to and from sub-process, messages, and events, we mainly use data associations. However, when models increase in complexity, the data associations may clutter the diagram. Thus, for trivial parameters, we use the data input and output sets in the non-graphical properties of the corresponding task.

An important BPMN feature which helped us model functionality that could be reused across different WSN applications is the encapsulation mechanism provided by sub-processes. The same encapsulation mechanism for BPMN pools would prove a useful mechanism for building a group hierarchy.

7.1.4 Limitations

There are also limitations to the modeling approach described in this thesis with respect to conventional programming languages. One such limitation is that it is not possible to change the binding between an event and an asynchronous operation once the latter has been started. Even though this is possible with conventional programming languages, such a style is prone to errors which might be difficult to trace.

From our practical experience, computationally intensive tasks such as averaging and filtering are created faster using conventional methods. These tasks should then be encapsulated and provided as building blocks, i.e., either as sub-processes or directly as an API method which can be reused in models.

Another aspect which is better controlled by conventional programming methods are low-level, hardware-specific operations and bit-level optimizations for an efficient and compact implementation. For example, packing multiple bits representing state in a byte array is cumbersome and at times not possible using modeling alone. By design, the underlying hardware is not accessible through modeling, which provides a higher abstraction. Thus, changing a single bit in the register of the radio, or implementing a software SPI driver is not possible using our methodology. However, we argue that the benefit of modeling at a higher abstraction level and the graphical overview of the entire application structure outweigh these limitations for typical application scenarios. Furthermore, the run-time platform can be optimized with respect to the hardware-specific parameters.

[3] We exclude the complex gateway which has no execution semantic (cf. Section 3.1.3).

7.2 Efficiency

We assess the efficiency of our modeling approach in terms of resource usage from two perspectives. On the one hand, we measure the effects of the proposed run-time and code generation optimizations for WSN applications. On the other hand, we measure the overall resource efficiency of the final result, i.e., we compare the applications generated from models with their manually coded equivalents. The resources we consider are energy and memory because they influence the economic viability of WSN applications.

7.2.1 Run-time

To show the effectiveness of the run-time platform abstractions presented in Section 6.2, we implemented them on the Mote Runner platform which combines VM and OS. In our case, the main goal of the optimizations is to counterbalance the overhead of a VM when compared with native implementations. We compare the total energy consumption of a Mote Runner implementation and a TinyOS implementation of the same application functionality.

For comparison, we selected a monitoring application under development at IBM [177]. The task of the application is to observe soil vibrations of an area and to determine their maximum velocity and frequency. Motes report the aggregated values of 100 bytes every 10 s to a base station and issue alerts in case certain thresholds are exceeded. The system has been designed and tested to run for several months or even years.

For energy consumption and sensing quality considerations, a node consists of an IRIS mote to run the network stack and a custom sensor board [177] featuring vibration sensors and a low-power FPGA for signal processing. The sensor board runs continuously to process the signal data, and periodically passes the computed vibration parameters to the IRIS mote. As the sensor board's power consumption is the same for both systems, our comparison focuses only on the IRIS mote running the network stack.

The networking stack is a hierarchical TDMA-based protocol with a central controller imprinting a routing tree and schedule onto the network [70]. Within a 10 s TDMA superframe, each network node has slots assigned to synchronize with its routing tree parent, to synchronize its children, to collect the children's messages, and to forward them to the parent, including its own data. The rest of the time is spent asleep.

The networking stack has been implemented by two independent teams for TinyOS and for Mote Runner to compare their performance. We measured the black-box energy consumption of these two implementations with the same code structure and approximately the same number of code lines. To analyze the total power consumption, we first measure the energy consumed in the sleep phase for the two implementations.

Sleep Phase (Application-Independent)

An important difference between TinyOS and Mote Runner is the number of wake-ups during the sleep phase, as shown in Figure 6.1. TinyOS wakes up in regular $T_{MAX} = 225$ ms intervals until the specified deadline. This behavior stems from a sleep strategy with a fixed pre-scaler and counter values. On the

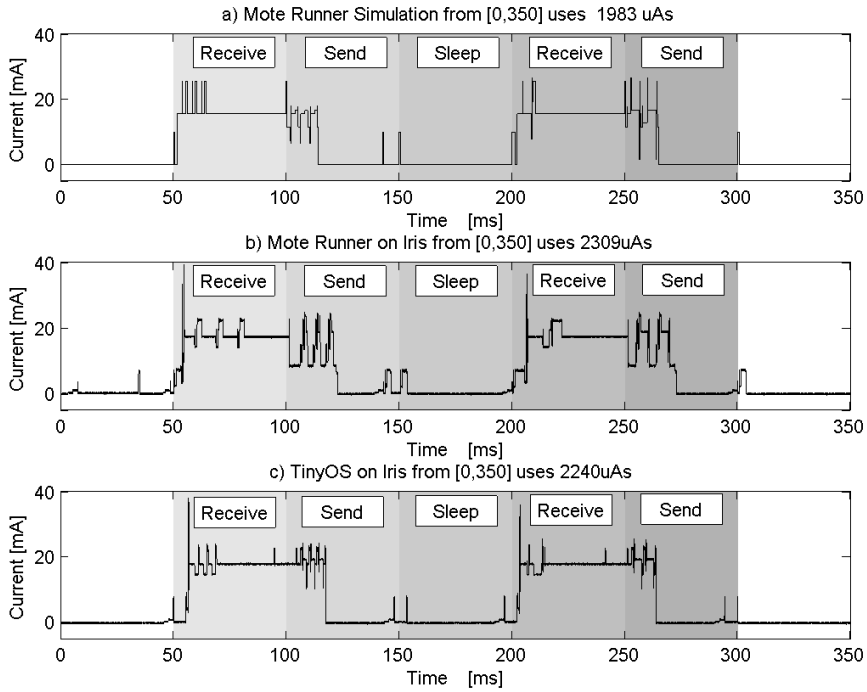


Figure 7.2: Application active-phase power trace comparison. The Mote Runner implementation uses a VM and in the active phase the application consumes only 4% more energy than the TinyOS implementation on the IRIS platform. The slower processing of managed code is compensated by using a radio API which micro-manages radio power states and transitions, thus reducing the overall energy consumption. Micro transitions are visible in the Mote Runner power trace, as there are more levels corresponding to more radio and MCU states than in the TinyOS power trace.

other hand, Mote Runner wakes up only once, sleeping for up to $T_{MAX} = 8\,\mathrm{s}$ based on our dynamic sleep strategy described in Section 6.2.1.

By inserting the above numbers in Equation (6.2), we obtain the ratio of wake-up to sleep energy for TinyOS, $\eta_{TOS} = 75\%$, and Mote Runner, $\eta_{MR} = 7.7\%$, respectively. In the case, in TinyOS almost ten times more energy is spent in the intermittent wake-ups than in the sleep periods.

Active Phase (Application-Specific)

The active phase represents the core functionality of a sensor network application. This part typically involves sending, receiving, or forwarding messages, and sampling and applying filtering functions on the sensor data.

Computations on top of a VM are by definition more expensive than native code. This price is paid for the benefits of virtual homogeneity, managed code, ease of debugging, and maintenance capabilities such as dynamically updating applications. Yet, for applications whose behavior is not clearly dominated by computations, an efficient VM implementation and smart sleep strategy can result in an overall solution whose overhead is well-controlled and which, from

an overall application perspective, can compete with native solutions.

Figure 7.2 shows the power consumption measured for the active phase within a superframe for the TinyOS and Mote Runner implementations running on the real IRIS hardware as well as for the Mote Runner simulation. The active phase includes 5 slots of 50 ms duration, during which the mote receives 3 synchronization messages from its parent, sends 3 synchronization messages to its children, then sleeps one slot. In the fourth slot, it collects one message from a child node and forwards it during the fifth slot to the parent, appending its own message. Afterwards, the mote sleeps for 10 s until the start of the first slot in the next superframe.

The overhead of VM computations is visible in the wider spikes in the first receive slot in Figure 7.2: Mote Runner requires more time for processing messages than TinyOS. This overhead is compensated by the radio API, which micro-manages power as described in Section 6.2.2.

Figure 7.2(a) shows the power trace for the monitoring application running in the Mote Runner simulation. Before a slot starts, the application decides whether to sleep, enable radio reception, or transmit messages during the respective slot. Peaks in the receive and send slots correspond to radio activity. The Mote Runner simulation runs the same VM/OS and run-time libraries as the hardware and executes the exact same application byte codes. This allows the simulation to emulate the application execution down to low-level hardware calls, such as memory copy routines, writing to flash, or radio chip control. It is therefore a close match to the hardware as illustrated in Figure 7.2(b) and gives the tendency for and an estimation of the overall power consumption. Simulated execution times are shorter than on the real IRIS hardware mainly because of the different cost of native routines.

The power consumption of the IRIS mote with the TinyOS implementation is shown in Figure 7.2(c). This is an example where power management of the radio module was coded by hand in TinyOS, whereas in Mote Runner it was done automatically according to the API described in Section 6.2.2. Prior to sending a message, TinyOS is usually in receive mode.

Mote Runner optimizes this behavior and automatically handles radio-chip state transitions to more efficient power states before the actual transmission. In the Mote Runner power traces, these optimizations are visible as there are more power levels than in TinyOS, in particular before the radio activity spikes in the send slots. The different power levels show that Mote Runner effectively uses multiple radio and MCU power states to reduce power consumption. For example, in the send phase ([100,150] ms), most of the time before sending messages is spent in the power level TRX_OFF, followed by a short period in PLL_ON.

The wider time span between messages in the Mote Runner power trace ([100-150] ms and [200-250] ms) stems from different APIs to timestamp sent messages than in TinyOS. This is an example where in Mote Runner, the programmer has to manually include the send time to the message before sending, thus requiring a sufficient message preparation time, which was set here to 5 ms. On the receiver side, the receive callback provides the exact time for the start of the reception. In contrast, TinyOS has a dedicated message type for synchronization, which reserves 4 bytes in a message, into which it writes a timestamp right upon sending the message to the transceiver. The receiver will timestamp it immediately upon reception. Our measurements show that

the two mechanisms are equivalent in terms of synchronization accuracy. Using only one message transmission as a synchronization point, we measured a synchronization error with a standard deviation of 49.4 μs between the sensor nodes at the beginning of each 10 s superframe. This is comparable to the error achieved by TinyOS [49] and close to the 20 ppm drift of the IRIS mote's clock source. Mote Runner can thus provide sufficient time accuracy to support more advanced protocols, such as FTSP [99], which can further improve synchronization.

In summary, Mote Runner needs more time for processing because of the VM overhead, but remains competitive by micro-managing power using the presented communication abstractions and sleep optimizations.

Total Energy Consumption and Life Time

The total energy consumption for an application in a superframe, i.e., the time span t during which the mote sleeps, transmits, and performs functional activity, is the sum of the energy consumption in the active (q_{active}) and the sleep (q_{sleep}) phase. In our concrete case, q_{active} corresponds to the measurements in Figure 7.2. We compute q_{sleep} using Equation 6.1 and the evaluation results from Section 7.2.1 for Mote Runner and TinyOS. Figure 7.3(a) shows our results for the total energy consumption for different superframe durations. For long superframes, the application-independent sleep energy dominates the application-specific active energy.

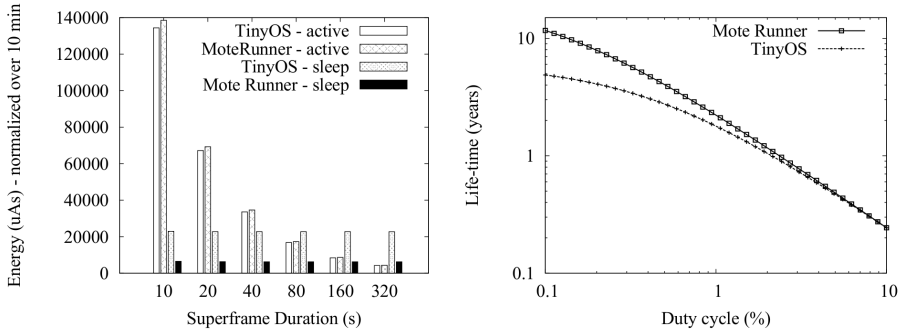


Figure 7.3: (a) Breakdown of total power consumption into sleep and active energy for the Mote Runner and TinyOS implementations for different superframe durations normalized over a period of 10 min — the active energy values are obtained from Figure 7.2. (b) Lifetime estimate for a mote powered by batteries (2000 mAh) and running the network stack implementation on Mote Runner and TinyOS for different duty cycles. To clearly separate the two estimates at higher duty cycles, both axes (duty cycle and lifetime) are logarithmic.

To put the power consumption into perspective, we compute the application lifetime, which depends on the charge consumed during one superframe. The lifetime $L(t)$ can be expressed as the fraction of the battery capacity C and the

sum of consumed charges over a superframe duration t as

$$L(t) = \frac{C}{(q_{sleep}(t) + q_{active})}. \tag{7.1}$$

We assume that the sensor nodes are powered by two off-the-shelf AA-sized lithium batteries with a capacity of 2000 mAh or 7.2 GC charge. Figure 7.3(b) plots the results for the expected lifetime of the Mote Runner and TinyOS implementations for the network stack under scrutiny. In this case, the break-even point where the Mote Runner sleep strategy pays off is for sleep periods of at least 2.5 s, corresponding to duty cycles lower than 10%. Thus, lower duty cycles will further improve the lifetime of the Mote Runner implementation in comparison to the TinyOS implementation.

7.2.2 Generation

Regarding the efficiency of the code generation, there are several aspects to consider. Firstly, our generated code implements the parallel semantic of BPMN models, akin to a multi-threaded program. Thus, for a first estimation of the overhead introduced through modeling, we quantify the cost of synchronization primitives. Secondly, timer and communication events are frequently used modeling constructs which introduce significant costs. Consequently, we quantify the improvement brought by reducing the number of timer objects based on patterns which exploit a time-based API (cf. Section 6.3.2). Finally, we quantify how an entirely generated application, including all code-generation optimizations (cf. Section 6.3), performs against a manually-coded equivalent with respect to the overall resource usage.

Our analysis focuses on flash and RAM usage (stack and heap), as well as energy consumption. We measured these quantities by monitoring the execution of the generated code based on both artificial models as well as models describing concrete applications (cf. Section 7.1.2). For our measurements, we used IRIS motes in the Mote Runner simulation environment which provides accurate estimates for memory and energy usage [23].

Cost of Synchronization

Gateways are an often used construct for merging logical, parallel-executing threads. Consequently, we evaluate the raw cost of synchronization mechanisms (AND-join and OR-join gateways), measured against the base case which does not implement any synchronization, i.e., executes through.

individual synchronization cost per incoming edge

We define the *individual cost* of merging as the amount of memory and energy resources required to implement the merge functionality for a single sequence flow incoming into a gateway. Using this definition, the individual cost does not depend on the number of incoming sequence flows. Thus, the individual cost of synchronization merge gateways does not depend on the complexity of the model.

Consequently, we can employ artificial models with a variable number (from 1 to 20) of gateways to compute the individual cost. Figure 7.4 shows the average individual costs for synchronization using `AND`-join and `OR`-join gateways. On average, the primitive energy cost of synchronization is 3 μAs for each incoming

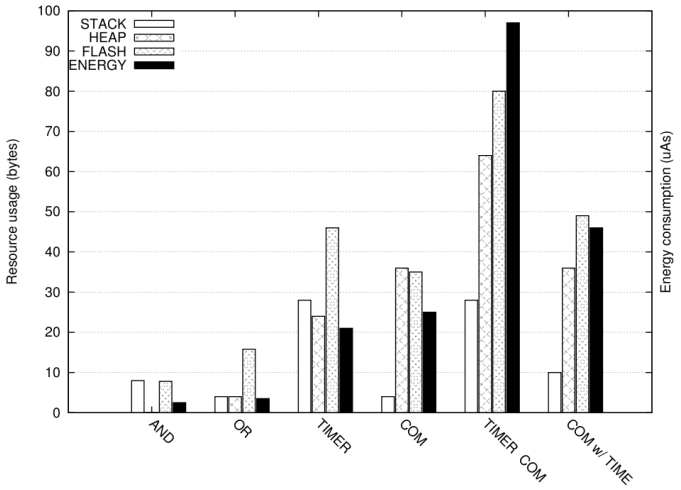


Figure 7.4: Average resource consumption for different primitives on the IRIS mote platform. The individual synchronization cost of each primitive (per gateway and per incoming sequence flow) is low when compared with the cost of timers and radio communication.

edge of a gateway. Moreover, each gateway requires only a fixed RAM cost of 8 bytes on average.

Note that the cost of OR gateways is twice that of AND gateways in terms of flash and energy usage. The main reason is that the synchronization of OR gateways requires non-local decisions using key-points which are marked on the execution paths at run-time. However, in many situations, OR gateways can be replaced by the more pedantic AND and XOR merge gateways at the cost of a larger graphical model.

total synchronization cost

However, the total synchronization cost in terms of flash, RAM, and energy usage is proportional to the number of gateways and the number of incoming sequence flows for each gateway. In addition, the cost depends on how many times a particular gateway has to merge. These parameters are application-specific and depend on the model description.

To estimate the total costs of synchronization, we use the frequency distribution of BPMN symbols provided by the study in [87], where a collection of more than 1200 BPMN models from academic education were analyzed. The largest model in the collection uses more that 400 symbols (based on the maximum number of nodes and highest density). The analysis confirms a previous study [191] on a smaller set of (more than 100) models from consulting, seminar education, and Web searches. Akin to natural languages, the ranked frequency distribution of BPMN symbols can be approximated by a simple, inverse power (= 1), Zipf distribution [191]. In other words, the symbols with rank k will appear in a model k times less than the symbol with the first rank.

Using the above observation and the statistical data provided in both empirical studies, we compute that AND-join gateways and OR-join gateways both represent on average 1.8% of the total symbols in a typical model. Furthermore,

using the statistics regarding concurrency in models provided by [87], we compute that the average (rounded up) number of incoming edges for each merge gateways is two.

As an example, assuming a large model with a total of 200 symbols, and according to the above distributions of symbols for typical BPMN models, we would find 4 AND-join and 4 OR-join gateways, each with two incoming edges. In this particular case, an estimate for the total cost of synchronization is: 128 bytes of RAM, 192 bytes of flash, and roughly 48 μA of spent energy. In the extreme case, for a theoretical model which only contains merge gateways, we are able to support more than 200 AND-joins or 200 OR-joins, given half of the memory resources (i.e., 4 kB RAM and 64 kB flash) of our target platform are fully available for applications. In both cases the limiting factor is the volatile memory.

In summary, for typical models, synchronization is a relatively cheap operation, which scales well for larger models.

Effect of Timers and Communication

By contrast, the individual raw cost of using timer objects (`TIMER`) and radio operations (`COM`) is significant. The individual costs are again independent of the application and can be measured using artificial models with a variable number (again from 1 to 20) of timers.

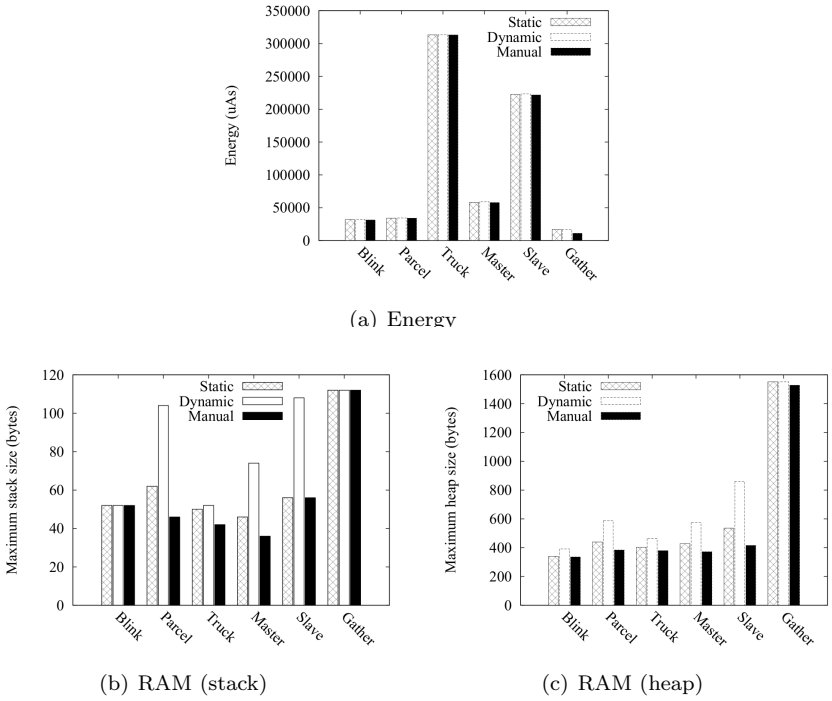
Our measurements in Figure 7.4 show that radio and timer operations cost almost three times as much as an incoming sequence flow for a gateway. Consequently, using timer objects for controlling communication (`TIMER & COM`) further increases costs. However, by directly using the time-based API for radio communication (`COM w/ TIME`), instead of timer objects, the resource usage can be reduced significantly. For each pattern (cf. Section 6.3.2), we gain 50% more RAM, 37% more flash, and reduce the energy consumption by half, when compared to using a dedicated timer object. These savings are considerable because timer events followed by communication activities represent a common pattern, particularly for sensor applications which employ TDMA-based schemes.

In practice, for our application scenarios (cf. Section 7.1.2), timer patterns accounted on average for 4.1% of the total model symbols. A similar distribution of timer and message events is also found in the previously described statistical studies [87, 191], which are based on a large collection of typical BPMN models. For our concrete scenarios, by using the time-based API, instead of individual timers, we reduced the RAM usage by 26% and the flash usage by 7% on average, for each individual application.

Overall efficiency

Finally, to quantify the overall efficiency of our approach, we measure resource usage of the generated code against the manually-written equivalents. To this end, we compiled each of the applications described in Section 7.1.2 once in static mode and once in dynamic mode, i.e., with all tasks treated as parallel tasks (c.f. Section 4.1.2). This is only possible because in none of our example application scenarios we encountered the need for this feature. Nevertheless, we think it is interesting to know the costs of supporting all BPMN features.

The total amount of consumed energy is an important efficiency measurement because the lifetime of battery-powered sensor nodes depends directly on it. The maximum stack size and the maximum heap size show how efficiently RAM resources are used, while the size of the binary does the same for the flash resources. Finally, the code size is a common indicator for code complexity. All measurements were obtained using the Mote Runner simulation environment running for 30 s, which is sufficiently long to measure and explore both the normal sequence flows as well as the exceptional code paths.



(a) Energy

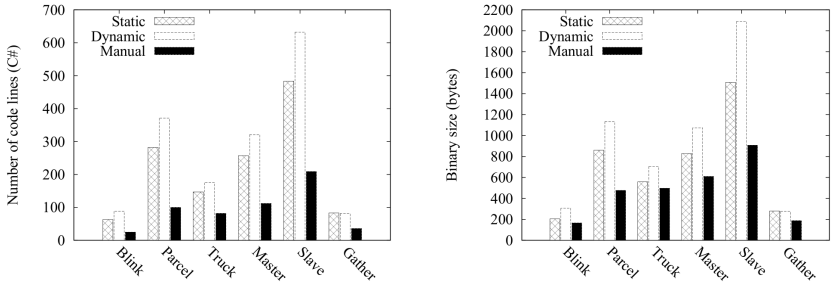
(b) RAM (stack)

(c) RAM (heap)

Figure 7.5: Comparison of generated code (static and dynamic) with the manual equivalent in terms of (a) energy and individual RAM usage for (b) stack and (c) heap.

Figure 7.5(a) shows the results of the energy consumption evaluation. The overhead for each compilation mode is only 1%. This is because WSN applications are reactive by nature, and sleeping periods dominate computations. However, the overhead for the RAM usage for both stack and heap averages at 10% in the static case and 50% in the dynamic mode as shown in Figure 7.5(c) and Figure 7.5(b), respectively.

In contrast, the size of the binary and the number of code lines show that the systematic transformation of the model results in twice more code than a software developer needs for the same task, as shown in Figure 7.6(a). On average, the overhead for the flash consumption is 44% for the static mode and three times more space for the dynamic mode as shown in Figure 7.6(b). The size of the dynamic BPMN run-time library is only 724 bytes and is included in the measurements for the dynamic case. The run-time library overhead in terms of stack and heap usage depends on the number of parallel process (and



(a) lines of code (b) flash

Figure 7.6: Comparison of generated code (static and dynamic) with the manual equivalent in terms of (a) number of lines of code and (b) flash footprint for the binary application.

is discussed in the next section).

In practice, the energy consumption is the most important efficiency characteristic of an application as long as the mote employed provides sufficient resources to host it. As we could show that the generated code fits on a typical mote and the additional energy consumption is very low, we argue that the benefits of the code generation outweigh the overhead introduced. This is especially true for the static mode, for which the overhead is moderate given that flash space is usually not the main limiting resource on motes.

Once a process behaved in the simulation as expected, we used IRIS [106] sensor nodes, which have an 8-bit micro-controller, 128 kB of flash and 8 kB of RAM, for a test deployment. We did not measure the resource consumption on the real hardware because the Mote Runner simulator provides accurate measurements by design. Instead, we only verified that the generated code is sufficiently efficient to be executed on such a resource-constrained device.

7.2.3 Break-even (Dynamic vs. Static)

The above measurements show that the dynamic mode, i.e., which uses the BPMN run-time, incurs considerable overhead in terms of space required for both persistent and volatile memory. The volatile memory overhead is mainly due to extra indirection layers, which increase the execution time. The extra layers stem from the inheritance mechanism and the BPMN run-time (cf. Section 4.4). For the above reasons, the dynamic mode will always require more RAM resources.

Compared with the equivalent static mode, the indirection in the dynamic mode influences the execution time, whose overhead is clearly exposed by the increased energy consumption. While the difference in energy consumption between the static and dynamic mode is on average below 2%, the increase in volatile memory usage is considerable. In some cases (e.g., *Parcel* and *Gather*), the dynamic mode requires approximately twice as much volatile memory. However, because functionality is shared, the dynamic mode can reduce the persistent memory consumption, for a large number of parallel instances.

Flash Footprint Break-event Point

For platforms in which enough volatile memory is available but in which the persistent memory is limited, an interesting question is: where is the break-even point in terms of the flash footprint? At the *break-event point*, using code generation in static mode is as expensive as the dynamic mode in terms of the flash footprint. One possible heuristic for finding the break-even point is to generate code using both modes, and, subsequently, comparing the flash space for the static mode with the flash space of the dynamic mode, which includes the BPMN run-time library. Then, the generated code which uses less persistent memory is selected. This comparison is only possible if the number of parallel instances is known at compile time or if the model is static (cf. Section 4.1.2).

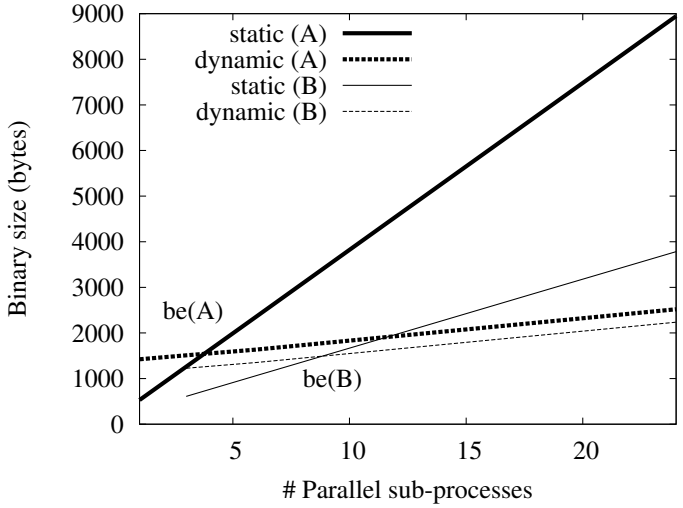


Figure 7.7: Two break-even points (`be`) when the `dynamic` code (including BPMN run-time) uses as much persistent memory (y-axis) as the `static` code. The break-even point depends on both the number of parallel sub-processes (x-axis) and the individual sizes of the sub-process (`A`,`B`).

Figure 7.2.3 shows the break-even point for two cases for a process using multiple instances of the same sub-process. The x-axis shows the number of parallel instances, whereas the y-axis shows the flash space required by the resulting application. For the dynamic mode, the flash size (724 bytes) of the BPMN run-time is included in the measurements. In case `A`, an artificial model starts (using a parallel gateway) multiple instances of a medium-sized sub-process, namely, 411 bytes (of flash) in static mode and 553 in dynamic mode. For case `B`, a similar parallel model uses a smaller sub-process, namely, 193 bytes (of flash) in static mode, and 269 in dynamic mode. In both cases, events are thrown in the sub-process which are caught either on the border or by using special event sub-processes. The functionality of event matching is duplicated by the static

code, whereas it is shared in the dynamic code.

The break-even point where the static code requires less persistent space than the dynamic code varies with both the size of each sub-process instance as well as the number of instances. Thus, for larger sub-processes, the savings from common functionality implemented by the dynamic code already pay off for a small number of parallel instances. Conversely, if sub-processes are smaller in size, the static code requires significantly less flash space for up to ten parallel instances. For our target platform and application domain ten parallel tasks represents a relatively large number, which we did not encounter in our example applications.

A hybrid approach, in which parts of the generated code use the dynamic BPMN library and parts are generated using the static method, is also possible. However, even if only a small part of a model requires the BPMN library to manage the dynamic behavior, an overhead is incurred. If, based on the requirements (e.g., limited data ranges), the model cannot exhibit dynamic behavior in practice, the best solution is to generate fully static code. The compiler parallelism warning can be overridden in this case. If parallelism can indeed not occur, we also recommend to re-factor the model diagram, e.g., by using gateways for synchronization or sequentializing tasks (cf. Section 3.2.5). In this way, the intention of a static model is expressed explicitly.

In summary, for typical applications, the static compilation mode is more efficient in terms of resource usage. However, the dynamic mode offers more generality, does not require prior knowledge of the process behavior, and is applicable to more complex models with a larger number of parallel instances.

7.3 A Note on Usability

Usability is an important aspect of any methodology or set of tools. In this respect, reusable components, rich libraries, good documentation, extensive tutorials, as well as both simple and complex examples play a crucial role. Thus, in addition to the efficiency measurements, we conducted a first series of experiments to assess the usability of our BPMN-based methodology based on our implementation of the tools. As the number of users is not statistically relevant, we limit ourselves to summarize some key aspects of our usability study.

The experiments involved both BPMN experts and students without prior knowledge of business process modeling. The participants without prior knowledge were given a one-hour introduction to BPMN. From November 12th to December 1st 2011, two technical users and three BPMN experts from the IBM Research - Zürich Laboratory as well as seven technical users (students) from the Swiss Federal Institute of Technology Zürich (ETHZ) participated in our study. The entire infrastructure for the experiment ran on a PC with a peak load of seven concurrent users. Both groups used our Web-based tools to solve a set of three tasks, with increasing degrees of difficulty. Users could immediately verify the behavior of their models using the Mote Runner simulation environment and dashboard.

The time allocated to solve all tasks was one hour. The first task involved a simple periodic behavior. In the second task, different synchronization primitives were used to actuate multiple LEDs in parallel. The last task involved a typical monitoring scenario in which a sensor node checked that the tempera-

ture value stayed between two thresholds and transmitted the current reading to a gateway. A value above the upper threshold was signaled with a red LED, whereas a value below the lower threshold was signaled with a yellow LED. The threshold values could be dynamically configured by a corresponding message from the gateway.

In general, all participants found the tools and integration with the simulation environment intuitive to use. To simplify the effort required from users, actuation and sensing tasks were encapsulated in reusable sub-processes. For the participants without prior knowledge of BPMN, some confusion was caused by the different flow notation for control, communication, and data. Users with a programming background requested more guidance and better documentation to help them assign the inputs and outputs to tasks and sub-processes.

Nevertheless, the technical users agreed that models did help understand and discuss the overall application. The feedback from users (and more than two dozen peers) with a business modeling background (including active members of the BPMN standardization body) acknowledged the strong point of our work: not changing the language or the semantics.

While all participants managed to easily solve the first two tasks, the last task was only solved by one quarter of the participants. Computationally intensive tasks posed a challenge for most participants. A hands-on tutorial for modeling with BPMN can help clarify some of the reported issues, such as the difference between the flow notations, and the semantics of inputs, outputs, and assignments.

The usability of the tools is an area for further improvement with more auto-completion and guidance features, more ready-to use examples and online documentation. In a sense, business process modeling should become more similar to the Eclipse IDE for Java programming. Furthermore, a rich set of libraries which implement standard and reusable computational and communication tasks are needed.

Summary

This chapter evaluated our methodology in terms of efficiency and ability to express WSN applications. On the one hand, we are able to express typical WSN applications using our BPMN-based methodology. On the other hand, our evaluation results showed that the efficiency criteria were fully met in terms of both energy and memory usage. Thus, the logic of business processes can be embedded further into the fabric of the environment using economically viable WSN platforms. However, the implementation of the integrated tools is not yet perfect in terms of usability. Consequently, the focus of further research should be on training both communities as well as improving the usability of the tools.

Chapter 8

Conclusion and Future Directions

In the following, we first summarize the results and contributions of this thesis. We then present our conclusions before discussing future research directions.

8.1 Summary of Results

This thesis presented a novel, model-based methodology which integrates WSN within business processes from an early phase and covers the most important phases in the BPM cycle. Our methodology starts with the design phase, continues to a generation-based implementation phase, which is followed by a simulation and testing phase before the final execution phase. This approach enables trained business experts as well as software developers to describe WSN applications at higher levels of abstraction using graphical models. Entire applications, which execute on resource-constrained sensor nodes, are then automatically generated from the model specifications.

In contrast to other modeling approaches in the WSN space, we used an open-industry standard and widely-used graphical language (BPMN) for describing applications. Furthermore, because we want the graphical models to be understood by both a business and an IT audience, we maintained the purity of the language chosen. In this sense, we explicitly refrained from syntax and semantic modifications to BPMN.

Consequently, this thesis introduced a modeling style for BPMN which allows one to express the reactive, distributed, and real-time requirements of WSN applications. Furthermore, we described a pattern-based compilation method for transforming BPMN models to event-based code. Our methodology considered all aspects of process models, that is, not only the control flow but also communication, data, and events. Furthermore, we showed how the synchronization and different event cancellation semantics of BPMN can be implemented both as an efficient, fully static program or by using a dynamic run-time. For the dynamic case, which is required when multiple parallel instances of the same sub-process execute concurrently, we introduced a thin BPMN run-time.

Moreover, this thesis focused on the efficient usage of resources to meet the target platform requirements and offer a commercially viable solution. Our

target platform is the class of tiny WSN nodes which use 8-bit MCUs with as little as 128 kB flash and 8 kB RAM, and operate on batteries. Well-established solutions from the business domain, such as process engines, Web Services, and XML-based process execution languages (e.g., BPEL), are not a suitable option in our case. This is because the classical business solutions introduce dependencies to monolithic middleware, which require significant resources that WSN nodes do not offer.

To reduce the energy and memory usage, we quantified the cost of individual operations (e.g., sleeping, radio communication) and the cost of primitives used for code generation, (e.g., synchronization code, timers, and buffers). Subsequently, we introduced a set of correlated optimizations which affect both the underlying WSN run-time platform and the code generation steps. In this way, we were able to generate applications that can compete with manually-coded equivalents in terms of resource usage.

Using a combination of practical and simulated measurements, this thesis showed that an efficient execution of business processes is possible even on resource-constrained WSN platforms. On average, the applications generated using our methodology use 10% more RAM and 44% more flash than their manually-coded equivalents. Furthermore, for typical low-duty cycle WSN applications, such as data-gathering and monitoring, the generated code uses only 1% more energy than manually written code. This is because such applications typically sleep more than 90% of the time. For such applications, our proposed sleep optimizations and automatic micro-management approach clearly payed off. We were able to not only compete with but even outperform existing native platforms for duty cycles below 10%. However, the overhead incurred for high-duty cycle applications for our system depends on the complexity of the computations.

8.2 Conclusion

The benefits of modeling WSN applications are a higher level of abstraction, a better overview of business process, an aligned implementation through code generation, and an integration by design. In general, these benefits come at a price in terms of efficiency and resource usage. However, we argue that the benefits outweigh the costs incurred. Organizations are thus quicker in adapting their WSN deployments to match new requirements for business processes which need to control and interact with the real environment.

Models for business processes facilitate the discussion between different participants from the business as well as the engineering sides. Having a common, in this case graphical, language facilitates reaching an agreement and validating both technical and business requirements. Our usability study is merely a starting point aimed at assessing the ease of adoption of such model-based, process-centric development methods. Even though the feedback from users familiar with BPMN is positive, the study suggests that further improvements of the integrated modeling tools are required for a wider adoption.

Generating applications from models does not mean that traditional software and hardware development will become obsolete. Innovations and optimizations in technical disciplines are still required to provide businesses with new platforms that expand the offering of products and services, as shown by the different

energy management optimizations presented in this thesis. In other words, a technical understanding is complementary to executable and efficient business processes.

Compared with conventional methods, graphical models have some inherent limitations with respect to low-level operations. For example, computationally intensive tasks, such as applying filters or performing a frequency analysis, are easier to implement in software or even directly in hardware. Once implemented, such tasks should be exposed as building blocks which can be reused in process models. The alternative approach is to model such tasks exhaustively from primitive elements, which is possible but cumbersome in practice.

8.3 Outlook

In a general sense, our integrated set of tools enable trained professionals to become "embedded programmers" using graphical models. An interesting path to explore is whether our modeling approach can be successfully applied to even larger audiences than the business and technical users discussed in this thesis. Would domain-specific scientists or even people responsible for administrative tasks be able to use the same graphical modeling tools? To answer this question, usability and comparative studies on a larger scale are required. One can even go a step further, e.g., to formally show what kind of equivalence exists between such executable business process models and conventional programming languages.

A large body of applications and software already exists. However in the near future these will require changes. If these changes are significant, it might prove cheaper in some cases to completely re-implement an entire solution. Instead, what if existing applications, which also provide source code, could be automatically imported into modeling tools? How about applications that do not provide the source code but whose external behavior (e.g., communication, actuation) could be observed and analyzed. Could these also be converted to models? In this case, the model could be used to perform the changes and regenerate a new application. This would enable not only the use of existing software as models, but also the updating of the models based on post-generation alterations performed in source code. The main challenge here is that the graphical representation may be too large for practical purposes. Thus, techniques that are able to visualize, re-factor, and even compact the resulting models are required.

Further, more practical, directions of our research could explore and compare different underlying platforms in terms of resource usage. This can be achieved, for example, by enabling different compiler back-ends for other WSN platforms. Conversely, the modeling tools should be adapted to allow the import of the corresponding API for the different platforms. To better support the testing and simulation of models, a substantial advancement in debugging distributed models is required. For example, the ability to specify distributed assertions as well as distributed conditions and watch points can facilitate the verification of the behavior of embedded processes.

Another interesting aspect is to extend the BPMN standard to include a hierarchy concept for entities (or pools). A hierarchy of entities allows a network to be structured into groups which can be further subdivided. The direct benefits are a simplification of model descriptions. Furthermore, a hierarchy of

entities would also facilitate modeling different macro-programing abstractions based on groups of logical neighborhoods.

Going a step further, is our methodology applicable to other computing platforms? Example platforms are not only conventional PCs but also portable computers such as sub-laptops, PDAs, and mobile phones. On the one hand, in terms of code generation, scaling-up offers more resources. In this case, the code generation would have to account for thread-based environments instead of the event-based execution discussed in this thesis. Model-driven (technical) solutions already provide solutions for platforms with sufficient resources and thread-based environments. Nevertheless, if the goal is a custom, minimal business process engine, then our methodology is applicable. On the other hand, scaling-up our methodology to the large back-end servers of business infrastructure is less reasonable. The focus of such central infrastructure systems is different and established solutions based on process engines running on application servers exist. Such systems are designed to manage the execution of parallel processes, offering Internet-accessible services with support for multiple concurrent users.

A different scale-up question refers to the modeling itself. Imagine a situation in which an interactive process must be tailored to specific customer needs on-demand and at the customer site. Thus, instead of requiring a conventional PC for design, development, and deployment, the business process may be modeled, compiled, and distributed in a WSN all from a portable device such as a mobile phone. The challenges here are multiple. Can models be entirely compiled on portable devices? Moreover, can models be visualized, browsed, and edited using screens with limited resolutions? Also, would the WSN management tools, which are required for deployment, fit the requirements of portable devices?

Following Moore's law, hardware platforms are constantly becoming cheaper, smaller, faster, and consume less energy. This enables our model-based methodology to be applied to an even larger range of embedded hardware. Furthermore, the platform-specific optimizations proposed in this thesis may already be integrated into the underlying systems. With advances in technology, businesses can design new processes which describe innovative and pervasive applications, which can be tailored for individual customers. However, small, embedded 8-bit platforms will always stay around; they will get smaller, cheaper, and even more prevalent. Thus, the work presented in this thesis remains relevant already in this domain.

Chapter 9

Annex

For completeness, we include in the annex some example models which were developed using our tools. We also include snippets of the JSON format used as input for the compiler as well as excerpts from the generated source code.

9.1 Example Models

The following two models show the details for the behavior of the TDMA protocol described in Section 7.1.2.

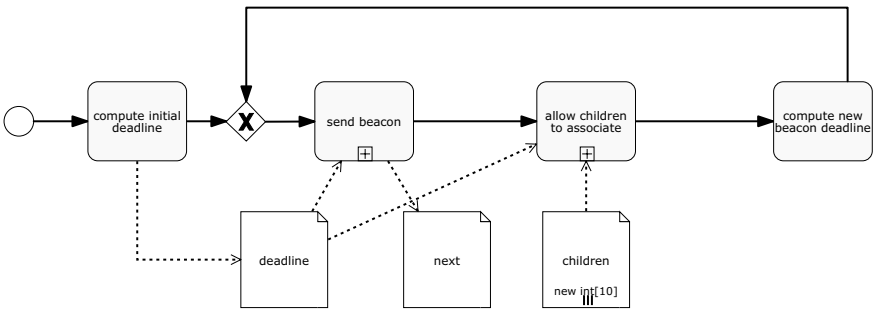


Figure 9.1: Top-level process controlling the `Master` in a simple TDMA-based multi-hop protocol.

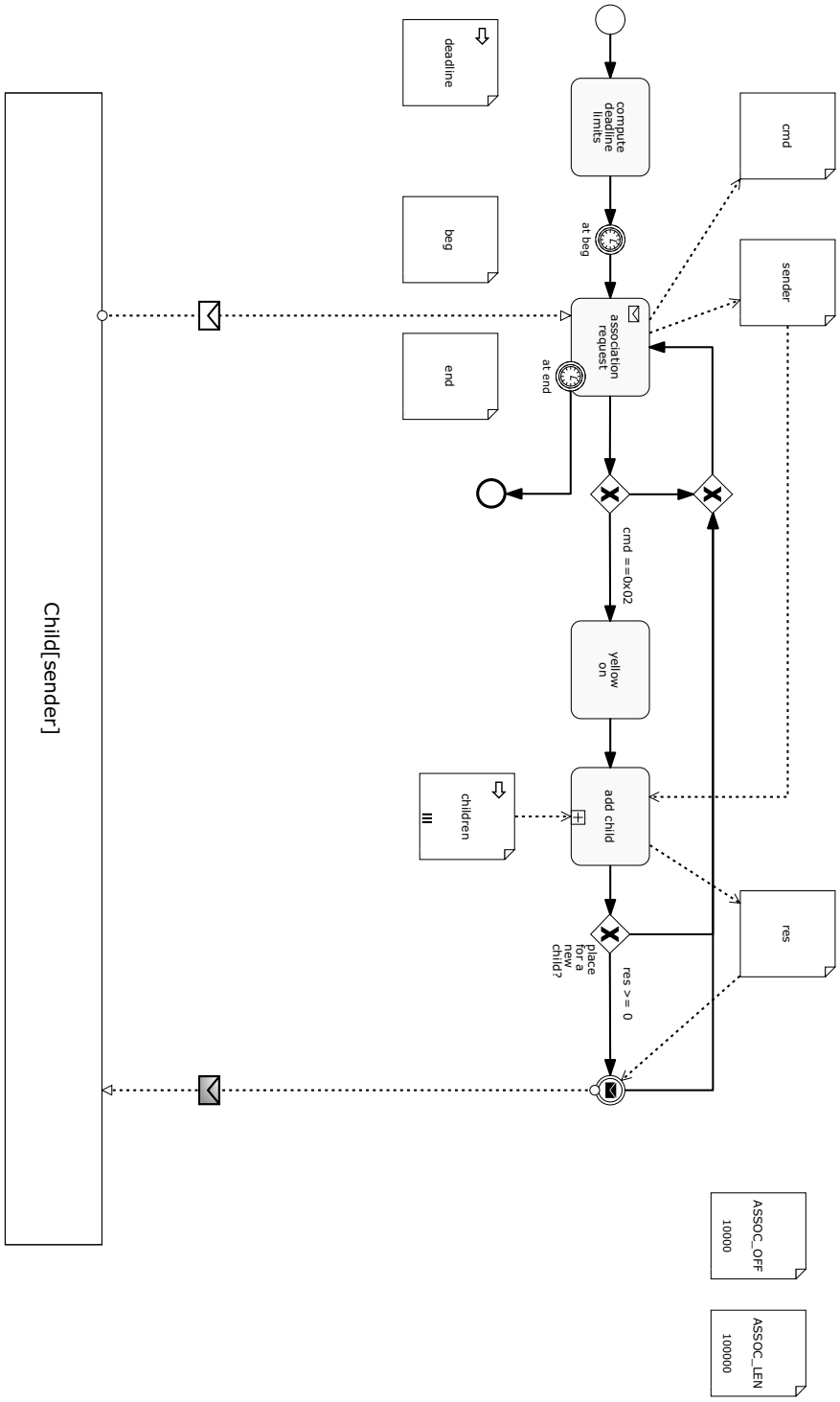


Figure 9.2: Reusable model which describes how children associate to a parent.

9.2 Input Format

The following listing represents a small extract of a serialized model exported from the Oryx BPMN editor. The model is encoded using JSON which our compiler tool accepts as input. Apart from the information relevant for code generation, such as inputs and outputs, stencil type, or unique identifiers, the serialization also includes the position of graphical elements in the diagram.

```
...
  {
      "bounds": {
          "lowerRight": {
              "x": 430,
              "y": 346
          },
          "upperLeft": {
              "x": 330,
              "y": 266
          }
      },
      "childShapes": [],
      "dockers": [],
      "outgoing": [
          {"resourceId": "oryx_97A389EC-3D7D-4A66-9213-DBFA1E1E4288"},
          {"resourceId": "oryx_81C98D53-95DB-46C1-8677-3FB8E00989BB"}
      ],
      "properties": {
          "activitytype": "Sub-Process",
          "adhoccancelremaininginstances": true,
          "adhoccompletioncondition": "",
          "adhocordering": "Sequential",
          "assignments": "",
          "auditing": "",
          "behavior": "all",
          "bgcolor": "#ffffcc",
          "callacitivity": "",
          "categories": "",
          "completioncondition": "",
          "completionquantity": 1,
          "complexbehaviordefinition": "",
          "datainputset": {
            "items": [{
              "isCollection": "false",
              "itemKind": "Information",
              "name": "no",
              "structure": "byte",
              "whileexecuting": "false"
            }],
            "totalCount": 1
          },
          "dataoutputset": {
            "items": [{
              "isCollection": "false",
              "itemKind": "Information",
              "name": "output",
              "structure": "uint",
              "whileexecuting": "false"
            }],
            "totalCount": 1
          },
          "diagramref": "",
          "documentation": "",
...
      },
      "resourceId": "oryx_63217D95-245E-4245-BFAF-04DB0E698131",
      "stencil": {"id": "CollapsedSubprocess"}
  },
...
```

9.3 Generated Source Code

The following listing shows a snippet of the C# code which is generated for the `Master` node from the above TDMA protocol models. The code was generated using the dynamic compilation mode (cf. Section 4.4).

```
// ------------------------------------------------------------
// =       Automatically generated using bxc compiler         =
// ------------------------------------------------------------
namespace master
{
  using com.ibm.saguaro.system;
  using com.ibm.saguaro.bpmn;


  // -------------------------------------------
  class AllowAssociation : AsyncTask{
    long deadline;
    long end;
    int res;
    long ASSOC_LEN = 100000;
    int[] children;
    byte cmd;
    long ASSOC_OFF = 10000;
    int sender;
    long beg;
    static long __rxForeverTicks = 0xFFFFFF;

    //
    private void __rxForever(uint info)
    {
      // <BEGIN> API specific: receive forever;
      Radio.enableRx(Time.currentTicks()+__rxForeverTicks);
      // <END> API specific: receive forever;
    }

    // association request
    private void msg_1_handler(byte[] pdu, uint len, long time, uint quality)
    {
      // <BEGIN> API specific RECEIVE unpack;
      this.cmd = pdu[9];
      sender = (int)Util.get16le(pdu,7) /* <SPECIAL source> */;
      // <END> API specific RECEIVE unpack;
      // <BEG> API specific RECEIVE behaviour;
      Radio.cancel();
      // <END> API specific RECEIVE behaviour;
      if (cmd ==0x02 )
      {
        LED.setState(0, 1); // yellow on
        int res = Master.AddChild(children, sender); // add child
        if (res >= 0 )
        {
          // <BEGIN> API specific send - pack variables;
          Master.msg[0] = 0x61 /* <set message type> */;
          Master.msg[1] = 0x88 /* <inc sequence number> */;
          Master.msg[2]++ /* <inc sequence number> */;
          Util.set16le(Master.msg,3,44656);
          Util.set16le(Master.msg,7,Master.shortAddress);
          Util.set16le(Master.msg,5,(uint)sender);
          Master.msg[9] = 0x12;
          Util.set16be(Master.msg,10,(uint)res);
          Radio.transmit(0, Master.msg, 0, 12, msg_2_handler);
          // <END> API specific send - pack variables;
        }
        else
        {
          // <BEGIN> API specific: wait forever until we receive one message;
          Radio.setShortAddr(Master.shortAddress);
          Radio.setRxDone(msg_1_handler_rxd);
          Radio.setRxHandler(0,msg_1_handler);
          Radio.enableRx(0,beg,end);
          // <END> API specific: wait forever until we receive one message;
        }
```

```
      }
      else
      {
        // <BEGIN> API specific: wait forever until we receive one message;
        Radio.setShortAddr(Master.shortAddress);
        Radio.setRxDone(msg_1_handler_rxd);
        Radio.setRxHandler(0,msg_1_handler);
        Radio.enableRx(0,beg,end);
        // <END> API specific: wait forever until we receive one message;
      }
    }

    // at end
    private void msg_1_handler_rxd(uint info)
    {
      // save energy;
      Radio.sleep();

      finished(null);
    }

    //
    private void msg_2_handler(byte[] pdu, uint len, int status, long txend)
    {
      // <BEGIN> API specific: wait forever until we receive one message;
      Radio.setShortAddr(Master.shortAddress);
      Radio.setRxDone(msg_1_handler_rxd);
      Radio.setRxHandler(0,msg_1_handler);
      Radio.enableRx(0,beg,end);
      // <END> API specific: wait forever until we receive one message;
    }

    //
    public void start(long deadline, int[] children)
    {
      // <BEG> PARAMETERS;
      this.deadline = deadline;
      this.children = children;
      // <END> PARAMETERS;
      // compute deadline limits;
      beg = deadline + ASSOC_OFF;
      end = deadline + ASSOC_LEN;
      // <BEGIN> API specific: wait forever until we receive one message;
      Radio.setShortAddr(Master.shortAddress);
      Radio.setRxDone(msg_1_handler_rxd);
      Radio.setRxHandler(0,msg_1_handler);
      Radio.enableRx(0,beg,end);
      // <END> API specific: wait forever until we receive one message;
    }

    //
    public  AllowAssociation(AsyncTask parent) : base(parent)
    {
    }
...
```

Bibliography

[1] S. Adam. The wealth of nations. In E. Cannan, editor, *An Inquiry into the Nature and Causes of the Wealth of Nations*. Methuen & Co., Ltd., London, 5th edition, 1904. First published 1776, available online `econlib.org/library/Smith/smWN1.html`.

[2] K. Ahrens, I. Eveslage, J. Fischer, F. Kühnlenz, and D. Weber. The challenges of using SDL for the development of wireless sensor networks. In R. Reed, A. Bilgic, and R. Gotzhein, editors, *SDL 2009: Design for Motes and Mobiles*, volume 5719 of *Lecture Notes in Computer Science*, pages 200–221. Springer, 2009.

[3] Y. L. Antonucci, M. Bariff, T. Benedict, B. Champlin, B. D. Downing, J. Franzen, D. J. Madison, S. Lusk, A. Spanyi, M. Treat, J. L. Zhao, and R. L. Raschke. *Business Process Management Common Body Of Knowledge*. CreateSpace, 2nd edition, 2009.

[4] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. Uzmi. Optimized Java binary and virtual machine for tiny motes. In R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, editors, *Distributed Computing in Sensor Systems*, volume 6131 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2010.

[5] Atmel. Atmega640/1280/1281/2560/2561 preliminary. `atmel.com/dyn/resources/prod_documents/doc2549.pdf`, 2009.

[6] Atmel. Low power 2.4 GHz radio transceiver for ZigBee and IEEE 802.15.4 applications - AT86RF230. `atmel.com/dyn/resources/prod_documents/doc5131.pdf`, 2009.

[7] A. Bachir, M. Dohler, T. Watteyne, and K. K. Leung. MAC essentials for wireless sensor networks. *IEEE Communications Surveys and Tutorials*, 12(2):222–248, 2010.

[8] L. Bai, R. Dick, and P. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, IPSN'09, pages 85–96. IEEE, 2009.

[9] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the Workshop on End-to-end,*

Sense-and-respond Systems, Applications and Services, EESR'05, pages 19–24. USENIX, 2005.

[10] L. Bencini, F. Chiti, G. Collodi, D. D. Palma, R. Fantacci, A. Manes, and G. Manes. Agricultural monitoring based on wireless sensor network technology: Real long life deployments for physiology and pathogens control. In *Proceedings of the 3rd International Conference on Sensor Technologies and Applications*, SENSORCOMM'09, pages 372–377. IEEE, 2009.

[11] A. Bernauer, K. Römer, S. Santini, and J. Ma. Threads2Events: An automatic code generation approach. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, HotEmNets'10, pages 1–5. ACM, 2010.

[12] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks*, IPSN'09, pages 265–276. IEEE, 2009.

[13] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, August 2005.

[14] M. Brambilla, P. Fraternali, and C. Vaca. A notation for supporting social business process modeling. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 88–102. Springer, 2011.

[15] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[16] F. P. Brooks, Jr. *The mythical man-month.* Addison-Wesley, anniversary edition, 1995.

[17] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich VM for the resource poor. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 169–182. ACM, 2009.

[18] N. Burri, R. Flury, S. Nellen, B. Sigg, P. Sommer, and R. Wattenhofer. YETI: an Eclipse plug-in for TinyOS 2.1. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 295–296. ACM, 2009.

[19] G. Byrne, D. Lubowe, and A. Blitz. Driving operational innovation using lean six sigma. `ibm.com/services/us/ceo/leansixsigma`.

[20] Y. Byun, B. Sanders, and C. Keum. Design patterns of communicating extended finite state machines in SDL. In *Proceedings of 8th Conference on Pattern Languages of Programs*, PLoP'01, 2001.

[21] A. Caracaş and A. Bernauer. Compiling business process models for sensor networks. In *Proceedings of 7th International Conference on Distributed Computing in Sensor Systems*, DCOSS'11, pages 1–8. IEEE, 2011.

[22] A. Caracaş, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote Runner: A multi-language virtual machine for small embedded devices. In *Proceedings of the 3rd International Conference on Sensor Technologies and Applications*, SENSORCOMM'09, pages 117–125. IEEE, 2009.

[23] A. Caracaş, C. Lombriser, Y. Pignolet, T. Kramp, T. Eirich, R. Adelsberger, and U. Hunkeler. Energy-Efficiency Through Micro-Managing Communication and Optimizing Sleep. In *Proceedings of the 8th Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON'11, pages 55–63. IEEE, 2011.

[24] A. Caracaş and T. Kramp. On the expressiveness of BPMN for modeling wireless sensor networks applications. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 16–30. Springer, 2011.

[25] M. Ceriotti, L. Mottola, G. Picco, A. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks*, IPSN'09, pages 277–288. IEEE, 2009.

[26] E. Cheong, E. A. Lee, and Y. Zhao. Viptos: a graphical development and simulation environment for TinyOS-based wireless sensor networks. Technical Report UCB/EECS-2006-15, Electrical Engineering and Computer Sciences University of California at Berkeley, February 2006.

[27] N. Costa, A. Pereira, and C. Serodio. Virtual machines applied to WSN's: The state-of-the-art and classification. In *Proceedings of the 2nd International Conference on Systems and Networks Communications*, ICSNC'07, page 50. IEEE, 2007.

[28] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of the International Workshop on Middleware for Sensor Networks*, MidSens'06, pages 43–48. ACM, 2006.

[29] T. Curran, G. Keller, and A. Ladd. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice-Hall, 1998.

[30] T. Davenpot and J. Short. The new industrial engineering: Information technology and business process redesign. *Sloan Management Review*, pages 11–27, Summer 1990.

[31] R. Davis and E. Brabnder. *ARIS Design Platform: Getting Started with BPM*. Springer, 1st edition, 2007.

[32] G. Decker, H. Overdick, and M. Weske. Oryx – an open modeling platform for the BPM community. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 382–385. Springer, 2008.

[33] G. Decker, W. Tscheschner, and J. Puchan. Migration von EPK zu BPMN. `signavio.com/images/stories/whitepapers/epk2bpmn-public.pdf`, 2009.

[34] L. Doldi. *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*. Wiley, 2003.

[35] M. Dumas, A. Grosskopf, T. Hettel, and M. Wynn. Semantics of standard process models with OR-joins. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2007.

[36] A. Dunkels, J. Eriksson, L. Mottola, T. Voigt, F. J. Oppermann, K. Römer, F. Casati, F. Daniel, G. P. Picco, S. Soi, S. Tranquillini, P. Valleri, S. Karnouskos, P. Spieß, and P. M. Montero. Application and programming survey. Technical Report D-1.1, European Project makeSense (grant agreement no: 258351), October 2010.

[37] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th International Conference on Local Computer Networks*, LCN'04, pages 455–462. IEEE, 2004.

[38] C. A. Ellis and G. J. Nutt. Office information systems and computer science. *ACM Computing Surveys*, 12(1):27–60, March 1980.

[39] J. Ellul and K. Martinez. Run-time compilation of bytecode in sensor networks. In *Proceedings of the 4th International Conference on Sensor Technologies and Applications*, SENSORCOMM'10, pages 133–138. IEEE, 2010.

[40] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[41] C. Favre and H. Völzer. Symbolic execution of acyclic workflow graphs. In R. Hull, J. Mendling, and S. Tai, editors, *Business Process Management*, volume 6336 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2010.

[42] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, December 1991.

[43] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technologies*, 2(2):115–150, 2002.

[44] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th Symposium on Operating System Design and Implementation*, OSDI'08. USENIX, 2008.

[45] J. Freund, B. Rücker, and T. Henninger. *Praxishandbuch BPMN*. Hanser, 2010.

[46] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May 1998.

[47] G. Fuchs, L. Büttner, C. Damm, M. Hansen, F. Heisig, T. Seyschab, and R. German. Demo: The acoowee-framework. In *Proceedings of 7th International Conference on Distributed Computing in Sensor Systems*, DCOSS'11, pages 1–2, 2011.

[48] G. Fuchs and R. German. UML2 activity diagram based programming of wireless sensor networks. In *Proceedings of the Workshop on Software Engineering for Sensor Network Applications*, SESENA'10, pages 8–13. ACM, 2010.

[49] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys'03, pages 138–149. ACM, 2003.

[50] B. Gfeller, H. Völzer, and G. Wilmsmann. Faster OR-join enactment for BPMN 2.0. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 31–43. Springer, 2011.

[51] M. Ghercioiu. A graphical programming approach to wireless sensor network nodes. In *Proceedings of Sensors for Industry Conference*, SIC'05. IEEE, 2005.

[52] N. Glombitza, M. Lipphardt, C. Werner, and S. Fischer. Using graphical process modeling for realizing SOA programming paradigms in sensor networks. In *Proceedings of the 6th International Conference on Wireless On-Demand Network Systems and Services*, WONS'09, pages 57–64. IEEE, 2009.

[53] N. Glombitza, D. Pfisterer, and S. Fischer. Integrating wireless sensor networks into Web service-based business processes. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens'09, pages 25–30. ACM, 2009.

[54] Generic Modeling Environment. `isis.vanderbilt.edu/projects/gme`.

[55] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 1–14. ACM, 2009.

[56] Google. V8 JavaScript Engine. `code.google.com/p/v8`.

[57] GRATIS II. `isis.vanderbilt.edu/projects/nest/gratis`.

[58] D. Guinard, C. Floerkemeier, and S. Sarma. Cloud computing, REST and mashups to simplify RFID application development and deployment. In *Proceedings of the 2nd International Workshop on the Web of Things*, WoT'11, pages 1–6. ACM, 2011.

[59] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the Internet of things to the Web of things: Resource-oriented architecture and best practices. In D. Uckelmann, M. Harrison, and F. Michahelles, editors, *Architecting the Internet of Things*, pages 97–129. Springer, 2011.

[60] T. Gunarathne, D. Premalal, T. Wijethilake, I. Kumara, and A. Kumar. BPEL-Mora: Lightweight embeddable extensible BPEL engine. In C. Pautasso, C. Bussler, M. Walliser, S. Brantschen, M. Calisti, and T. Hempfling, editors, *Emerging Web Services Technology*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 3–20. Birkhäuser, 2007.

[61] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman. Sliver: A BPEL workflow process execution engine for mobile devices. In A. Dan and W. Lamersdorf, editors, *Service-Oriented Computing – ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 503–508. Springer, 2006.

[62] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[63] S. Haller. The things in the internet of things. In *Poster at the Internet of Things*, IoT'10, 2010.

[64] M. Hammer. Reengineering work: Don't automate, obliterate. *Harvard Business Review*, pages 104–112, July-August 1990.

[65] R. Hauser. Automatic transformation from graphical process models to executable code. Technical Report 1177, Eidgenössische Technische Hochschule Zürich (ETHZ), 2010.

[66] R. Hauser and J. Koehler. Compiling process graphs into executable code. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 129–243. Springer, 2004.

[67] M. Havey. *Essential Business Process Modeling*. O'Reilly, 2005.

[68] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, 2000.

[69] C. Hoare. The varieties of programming language. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 351 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1989.

[70] U. Hunkeler, L. H. Truong, and A. Caracaş. Synchronizing nodes of a multi-hop network. IBM Corporation. US Patent 20110268139, 2010.

[71] IBM. Business Process Management. `ibm.com/software/info/bpm`.

[72] IBM. Mote Runner Project Website. `zurich.ibm.com/moterunner`.

[73] IBM. Rational Rhapsody. `ibm.com/software/awdtools/rhapsody`.

[74] IBM. *WebSphere Business Process Management Samples and Tutorials*.

[75] IBM. *Rational SDL and TTCN Suite 6.3: SDL Suite Getting Started*, 2009.

[76] IBM. *Rational SDL and TTCN Suite 6.3: User's Manual*, 2009.

[77] IEEE. Wireless medium access control (MAC) and physical layer (PHY) specifications for low rate wireless personal area networks (LR-WPANs), 2006.

[78] A. Ilic, T. Staake, and E. Fleisch. Using sensor information to reduce the carbon footprint of perishable goods. *IEEE Pervasive Computing*, 8(1):22–29, January-March 2009.

[79] M. Jansen-Vullers and M. Netjes. Business process simulation - a tool survey. In *Proceedings of Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 77–96, 2006.

[80] JBOSS Community. jBPM. `jboss.org/jbpm`.

[81] S. Karnouskos, D. Guinard, D. Savio, P. Spiess, O. Baecker, V. Trifa, and L. de Souza. Towards the real-time enterprise: Service-based integration of heterogeneous SOA-ready industrial devices with enterprise applications. In *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing*, INCOM'09, 2009.

[82] T. C. Karunaratna. Nondeterminator-3: A provably good data-race detector that runs in parallel. Master's thesis, Massachusetts Institute of Technology, 2005.

[83] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische prozeßmodellierung auf der grundlage ereignisgesteuerter prozeßketten (EPK). In A.-W. Scheer, editor, *Erschienen in der Reihe: Veröffentlichungen des Instituts für Wirtschaftsinformatik.*, Heft 89. Universität des Saarlandes, 1992.

[84] T. Kramp and T. Visegrady. Just-in-time compiling or dynamic compiling (e.g., compiling java bytecode on a virtual machine). IBM Corporation. US Patent 20110035735, 2010.

[85] S. Krishnamurthy and L. Lange. Distributed interactions with wireless sensors using TinySIP for hospital automation. In *Proceedings of the 6th International Conference on Pervasive Computing and Communications*, PERCOM'08, pages 269–275. IEEE, 2008.

[86] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[87] M. Kunze, A. Luebbe, M. Weidlich, and M. Weske. Towards understanding process modeling - the case of the BPM academic initiative. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 44–58. Springer, 2011.

[88] B. C. Kuo and M. F. Golnaraghi. *Automatic Control Systems*. Wiley, 2003.

[89] M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen. Rapid design and evaluation framework for wireless sensor networks. *Ad Hoc Networks*, 6(6):909–935, August 2008.

[90] LEGO. MINDSTORMS. mindstorms.lego.com.

[91] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'02, pages 85–95. ACM, 2002.

[92] P. Levis, L. N., M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys'03, pages 126–137. ACM, 2003.

[93] M. Lipphardt, J. Neumann, and C. Werner. Self-organizing service distribution. In *Proceedings of the 6th International Conference on Embedded Network Sensor Systems*, SenSys'08, pages 389–390. ACM, 2008.

[94] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 183–196. ACM, 2009.

[95] F. Losilla, C. Vicente-Chicote, B. Álvarez, A. Iborra, and P. Sánchez. Wireless sensor network application development: An architecture-centric MDE approach. In F. Oquendo, editor, *Software Architecture*, volume 4758 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2007.

[96] D. E. Mahling, N. Craven, and W. B. Croft. From office automation to intelligent workflow systems. *IEEE Expert: Intelligent Systems and Their Applications*, 10(3):41–47, June 1995.

[97] M. Marin-Perianu, T. Hofmeijer, and P. Havinga. Implementing business rules on sensor nodes. In *Conference on Emerging Technologies and Factory Automation*, ETFA'06, pages 292–299. IEEE, 2006.

[98] M. Marin-Perianu, N. Meratnia, P. Havinga, L. de Souza, J. Muller, P. Spiess, S. Haller, T. Riedel, C. Decker, and G. Stromberg. Decentralized enterprise systems: A multiplatform wireless sensor network approach. *IEEE Wireless Communications*, 14(6):57–66, 2007.

[99] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys'04, pages 39–49. ACM, 2004.

[100] MathWorks. MATLAB. `mathworks.com/products/matlab`.

[101] MathWorks. Real-time workshop embedded coder. `mathworks.com/products/rtwembedded`.

[102] MathWorks. *Simulink 7 Getting Started Guide*.

[103] MathWorks. Stateflow. `mathworks.com/products/stateflow`.

[104] T. Matsushita. *Expressive Power of Declarative Programming Languages*. PhD thesis, University of York, 1998.

[105] J. McCulloch, P. McCarthy, S. M. Guru, W. Peng, D. Hugo, and A. Terhorst. Wireless sensor network deployment for water use efficiency in irrigation. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks*, REALWSN'08, pages 46–50. ACM, 2008.

[106] Memsic. IRIS 2.4 GHz wireless measurement system. `memsic.com/products/wireless-sensor-networks/wireless-modules.html`.

[107] J. Mendling, K. Lassen, and U. Zdun. Transformation strategies between block-oriented and graph-oriented process modeling languages. *Multikonferenz Wirtschaftsinformatik*, 2:297–312, 2006.

[108] J. Mendling, G. Neumann, and M. Nüttgens. Towards workflow pattern support of event-driven process chains (EPC). In *Proceedings of the 2nd Workshop XML4BPM 2005*, pages 23–38, 2005.

[109] J. Mendling and W. M. P. van der Aalst. Formalization and verification of EPCs with OR-joins based on state and context. In J. Krogstie, A. Opdahl, and G. Sindre, editors, *Advanced Information Systems Engineering*, volume 4495 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2007.

[110] S. Meyer, K. Sperner, C. Magerkurth, and J. Pasquier. Towards modeling real-world aware business processes. In *Proceedings of 2nd International Workshop on the Web of Things*, WoT'11, 2011.

[111] J. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, 2003.

[112] J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163, 1993.

[113] MOFScript. `eclipse.org/gmt/mofscript`.

[114] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 43(3):1–51, April 2011.

[115] L. Mottola and G. P. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the 1st International Conference on Integrated Internet Ad Hoc and Sensor Networks*, InterSense'06. ACM, 2006.

[116] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri. A framework for modeling, simulation and automatic code generation of sensor network application. In *Proceedings of the 5th Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON'08. IEEE, 2008.

[117] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 41(3):145–158, March 2007.

[118] V. Naoumov and T. Gross. Simulation of large ad hoc networks. In *Proceedings of the 6th International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWIM'03, pages 50–57. ACM, 2003.

[119] National Instruments. LabVIEW. `ni.com/labview`.

[120] National Instruments. *LabVIEW Getting Started with LabVIEW*.

[121] National Instruments. LabVIEW Wireless Sensor Network (WSN) Pioneer - Under the Hood. `zone.ni.com/devzone/cda/tut/p/id/9006`.

[122] National Instruments. NI WSN-3212 4 Ch, 24-Bit, Programmable Thermocouple Input Node. `sine.ni.com/nips/cds/view/p/lang/en/nid/207089`.

[123] T. Naumowicz, B. Schröter, and J. Schiller. Prototyping a software factory for wireless sensor networks. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 369–370. ACM, 2009.

[124] OASIS. Web Services Business Process Execution Language Version 2.0. `oasis-open.org/committees/wsbpel`, 2007.

[125] Object Management Group (OMG). Model Driven Architecture (MDA): Success Stories. `omg.org/mda/products_success.htm`.

[126] Object Management Group (OMG). Universal Modeling Language (UML) 2.0. `uml.org`.

[127] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, June 2003.

[128] B. O'Flynn, R. Martinez-Catala, S. Harte, C. O'Mathuna, J. Cleary, C. Slater, F. Regan, D. Diamond, and H. Murphy. Smartcoast: A wireless sensor network for water quality monitoring. In *Proceedings of the 32nd Conference on Local Computer Networks*, LCN '07, pages 815–816. IEEE, 2007.

[129] OMG. Business Process Model and Notation (BPMN) 2.0. `bpmn.org`.

[130] Oracle. Business Process Management. `oracle.com/technetwork/middleware/bpm`.

[131] Oracle Labs. Sun SPOT World. `sunspotworld.com`.

[132] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Proceedings of 31st Conference on Local Computer Networks*, LCN'06, pages 641–648. IEEE, 2006.

[133] C. Ouyang, M. Dumas, S. Breutel, and A. H. M. ter Hofstede. Translating standard process models to BPEL. In E. Dubois and K. Pohl, editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 2006.

[134] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology*, 19(1):1–37, August 2009.

[135] Papyrus. Open Source Tool for Graphical UML2 Modelling. `papyrusuml.org`.

[136] A. Pathak and M. K. Gowda. Srijan: a graphical toolkit for sensor network macroprogramming. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE'09, pages 301–302. ACM, 2009.

[137] C. Pautasso. BPMN for REST. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 74–87. Springer, 2011.

[138] A. J. Perlis. Special feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, September 1982.

[139] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik der Universität Bonn, 1962.

[140] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks*, IPSN'05, pages 364–369. IEEE, 2005.

[141] F. Puhlmann and M. Weske. Investigations on soundness regarding lazy activities. In S. Dustdar, J. Fiadeiro, and A. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2006.

[142] A. Reinhardt and R. Steinmetz. Exploiting platform heterogeneity in wireless sensor networks for cooperative data processing. In *Proceedings of the 8th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, August 2009.

[143] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. S. Silver, B. Silverman, and Y. B. Kafai. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[144] C. Richardson. The Forrester Wave: Business process management suites, Q3 2010. Technical report, Forrester, 2010.

[145] K. Römer. Time synchronization in ad hoc networks. In *International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc'01, pages 173–182. ACM, 2001.

[146] U. Sammapun, J. Regehr, I. Lee, and O. Sokolsky. Runtime verification for wireless sensor network applications. In B. Finkbeiner, K. Havelund, G. Rosu, and O. Sokolsky, editors, *Dagstuhl Seminar Proceedings*, 07011, 2008.

[147] SAP. NetWeaver Business Process Management. `sap.com/platform/netweaver/components/sapnetweaverbpm`.

[148] A.-W. Scheer. ARIS-toolset: Von forschungs-prototypen zum produkt. *Informatik-Spektrum*, 19(2):71–78, 1996.

[149] A.-W. Scheer. *ARIS - Business Process Frameworks*. Springer, 2nd edition, 1998.

[150] T. Schreiter. xBPMN++ towards executability of BPMN: Data perspective and process instantiation. Master's thesis, Hasso Plattner Institute, 2008.

[151] S. Schäfer. Secure trade lane: A sensor network solution for more predictable and more secure container shipments. In *Companion to the 21st Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA'06, pages 839–845. ACM, 2006.

[152] R. Sen, G.-C. Roman, and C. Gill. CiAN: A workflow engine for MANETs. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2008.

[153] Z. Shelby and B. Carsten. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2009.

[154] Y. Shi, K. Casey, M. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[155] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys'04, pages 188–200. ACM, 2004.

[156] B. Silver. *BPMN Method and Style: A Levels-based Methodology for BPM Process Modeling and Improvement Using BPMN 2.0*. Cody-Cassidy Press, 2009.

[157] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE'06, pages 78–88. ACM, 2006.

[158] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks*, IPSN'09, pages 37–48. IEEE, 2009.

[159] K. Sperner, S. Meyer, and C. Magerkurth. Introducing entity-based concepts to business process modeling. In R. Dijkman, J. Hofstetter, J. Koehler, W. M. P. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 166–171. Springer, 2011.

[160] P. Spieß, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. de Souza, and V. Trifa. SOA-based integration of the internet of things in enterprise services. In *Proceedings of the International Conference on Web Services*, ICWS'09, pages 968–975. IEEE, 2009.

[161] P. Spieß, H. Vogt, and H. Jütting. Integrating sensor networks with business processes. In *Real-World Sensor Networks Workshop at ACM MobiSys*. ACM, 2006.

[162] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[163] F. W. Taylor. *The Principles of Scientific Management*. History of Economic Thought Books. McMaster University Archive for the History of Economic Thought, 1911.

[164] A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009.

[165] TinyOS. Community Forum. `tinyos.net`.

[166] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN'05, page 67. IEEE, 2005.

[167] H. L. Truong, C. Faulkner, and A. Stanford-Clark. Wireless monitoring and control using MQTT-S over ZigBee. In *Second European ZigBee Developers' Conference*, pages 24–25, 2008.

[168] W. M. P. van der Aalst and B. K. Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. *Information and Software Technology*, 50(3):131–159, February 2008.

[169] W. M. P. van der Aalst, J. Nakatumba, A. Rozinat, and N. Russell. Business process simulation. In J. vom Brocke and M. Rosemann, editors, *Handbook on Business Process Management*, volume 1 of *International Handbooks on Information Systems*, pages 313–338. Springer, 2010.

[170] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[171] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data and Knowledge Engineering*, 68(9):793–818, 2009.

[172] The VINT Project. *The ns Manual (formerly ns Notes and Documentation)*, 2010.

[173] G. Wagenknecht, D. Dietterle, J.-P. Ebert, and R. Kraemer. Transforming protocol specifications for wireless sensor networks into efficient embedded system implementations. In K. Römer, H. Karl, and F. Mattern, editors, *Wireless Sensor Networks*, volume 3868 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2006.

[174] T. Wallentine. *Cadena 2.0: nesC Tutorial. A guide to using Cadena to develop nesC/TinyOS applications*, 2007.

[175] M. Weidlich, G. Decker, A. Großkopf, and M. Weske. BPEL to BPMN: The myth of a straight-forward mapping. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5331 of *Lecture Notes in Computer Science*, pages 265–282. Springer, 2008.

[176] T. Weigold, T. Kramp, and P. Buhler. ePVM - an embeddable process virtual machine. In *Proceedings of the 31st International Computer Software and Applications Conference*, COMPSAC'07, pages 557–564. IEEE, 2007.

[177] B. Weiss, H. L. Truong, W. Schott, A. Munari, C. Lombriser, U. Hunkeler, and P. Chevillat. A power-efficient wireless sensor network for continuously monitoring seismic vibrations. In *Proceedings of the 8th Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, SECON'11, pages 37–45. IEEE, 2011.

[178] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 381–396. USENIX, 2006.

[179] M. Weske. *Business Process Management. Concepts, Languages, Architectures*. Springer, 2007.

[180] S. A. White. Using BPMN to model a BPEL process. `bptrends.com`, 2005.

[181] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell. On the suitability of BPMN for business process modelling. In S. Dustdar, J. Fiadeiro, and A. Sheth, editors, *Business Process*

Management, volume 4102 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2006.

[182] Workflow Management Coalition (WfMC). XPDL support and resources. `wfmc.org/xpdl.html`.

[183] World Wide Web Consortium (W3C). Web of Services. `w3.org/standards/webofservices`.

[184] M. Wynn, M. Dumas, C. Fidge, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Business process simulation for operational decision support. In A. H. M. ter Hofstede, B. Benatallah, and H.-Y. Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2008.

[185] F. Zhao and L. Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann, 2004.

[186] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, and F. Cao. Compiling business processes: Untangling unstructured loops in irreducible flow graphs. *International Journal of Web and Grid Services*, 2(1):68–91, February 2006.

[187] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, HPCA'07, pages 121–132. IEEE, 2007.

[188] ZigBee Alliance. Standards Overview. `zigbee.org`.

[189] H. Zimmermann. OSI reference model — the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

[190] M. zur Muehlen and D. T. Ho. Service process innovation: A case study of BPMN in practice. In *Proceedings of the 41st Hawaii International Conference on System Sciences*, HICSS'08, page 372. IEEE, 2008.

[191] M. zur Muehlen and J. Recker. How much language is enough? theoretical and practical use of the business process modeling notation. In Z. Bellahsène and M. Léonard, editors, *Advanced Information Systems Engineering*, volume 5074 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2008.