UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SIMULATING SENSOR NETWORKS

Duarte Almeida Vieira

MESTRADO EM INFORMÁTICA

2010

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SIMULATING SENSOR NETWORKS

Duarte Almeida Vieira

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Francisco Cipriano da Cunha Martins

MESTRADO EM INFORMÁTICA

2010

Agradecimentos

Em primeiro lugar, gostaria de agradecer a orientação desta tese ao Professor Doutor Francisco Martins, que me ensinou, guiou e aconselhou de forma singular. Agradeço os ensinamentos, a introdução à investigação científica, mas também a confiança, a amizade e a boa disposição. Agradeço ainda a paciência e a compreensão nas épocas em que interrompi a investigação por motivos profissionais. Muito obrigado por tudo.

Gostaria também de agradecer aos meus pais a oportunidade e incentivo a estudar e, especialmente, o à vontade e o apoio que sempre senti em escolher o meu caminho. Agradeço à minha namorada Júlia a confiança e optimismo que sempre me transmitiu como só ela sabe e consegue. Esta tese é também dela.

Não posso deixar de agradecer aos meus colegas de trabalho a compreensão que sempre demonstraram com a minha frequente indisponibilidade. Finalmente, agradeço aos colegas e amigos da Faculdade de Ciências da Universidade de Lisboa, em particular ao Tiago Cogumbreiro, pela amizade e pelas discussões frutíferas sobre o tema desta tese.

Resumo

Nos últimos anos, as redes de sensores sem fios conheceram um grande impulso em variadas áreas, nomeadamente na monitorização industrial e ambiental e, mais recentemente, na logística e noutas aplicações que envolvem processos de negócio e a chamada *Internet das Coisas e dos Serviços*. Contudo, e apesar dos avanços que se têm verificado tanto em termos de *hardware* como de *software*, estas redes são difíceis de programar, testar e instalar. A simulação de redes de sensores é frequentemente utilizada para testar e depurar aplicações para redes de sensores, pois permite testar a execução de das aplicações em ambientes virtuais.

Esta tese aborda um problema que diz respeito a testar estas redes através de simulação: a definição (manual) de modelos. A nossa abordagem aponta para a geração de modelos de simulação directamente a partir de aplicações redes de sensores, em particular, modelos para o simulador VisualSense criados a partir de aplicações escritas em Callas, uma linguagem de programação para as redes de sensores. Para tal, criamos uma ferramenta capaz de gerar modelos que é paramétrica pelos modelos de rede e modelos sensores da rede que se pretende modelar, e ainda por um conjunto extensível de parâmetros de simulação. As nossas experiências mostraram resultados encorajadores na simulação de redes de grande escala, uma vez que conseguimos executar simulações com até 5000 nós.

À medida que as redes de sensores sem fios começam a ser utilizadas em processos de negócio, a informação que recolhem do ambiente tem cada vez mais influência no decurso dos fluxos de trabalho associados aos processos de negócio. De um modo geral, os testes levados a cabo em fluxos de trabalho fazem uso de informação gravada em fluxos de trabalho executados previamente, tornando difícil testar o sistema como um todo. Em alternativa, e como uma segunda proposta desta tese, propomos testar fluxos de trabalho através da incorporação de resultados obtidos nas simulações das aplicações das redes de sensores. Além de cobrir os casos cobertos pela primeira abordagem, esta técnica permite testar novos fluxos de trabalho, bem como as mudanças ocorridas num determinado fluxo de trabalho por acontecimentos no ambiente.

Palavras-chave: redes de sensores, simulação, sistemas de gestão de fluxos de trabalho

Abstract

In recent years, Wireless Sensor Networks have gaining momentum in several fields, notably in industrial and environmental monitoring and, more recently, in logistics. However, and in spite of the advances in hardware and software, Wireless Sensor Networks are still hard to program, test, and deploy. Simulation is often used for testing and debugging sensor networks because they allow us to perform deployments in virtual environments.

This paper addresses a key problem of testing such networks using simulation: (manual) model definition. Our approach is to generate simulation models directly from WSN applications, in particular, VisualSense simulator models from applications written in Callas, a programming language for WSN. For that purpose, we create a model generator tool that is parameterisable by network and sensor templates, and by an extensible set of simulation parameters. Our experiments show encouraging results on simulating large scale networks, as we are able to handle WSN with as many as 5000 nodes.

As Wireless Sensor Networks begin to play some role in business processes, the information they gather from the environment influences the execution of workflows. Generally, the tests carried out on these systems make use of recorded information in earlier workflow executions, making it difficult to test the system as a whole. Alternatively, and as a second proposal of this thesis, we propose testing such workflows by incorporating results obtained from the simulation of sensor network applications. Besides covering the situations described in the first approach, this technique allows the testing of new workflows, as well as the changes made to a given workflow by events in the environment.

Keywords: sensor networks, simulation, workflow management systems

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

2 Wireless Sensor Networks **5**
2.1 Applications 5
2.1.1 Current Applications 6
2.1.2 Envisioned Applications 7
2.2 Sensor Devices 7
2.2.1 Nanosensors 9
2.3 Communication 10
2.3.1 Protocols 10
2.3.2 Standards and Technologies 12
2.3.3 Gathering 13
2.4 Programming Wireless Sensor Networks 13
2.4.1 Operating Systems 14
2.4.2 Programming Models and Languages 14
2.5 Research Topics 16

3 Wireless Sensor Network Simulation **19**
3.1 Wireless Sensor Network Simulators 19
3.2 The VisualSense Simulator 22
3.2.1 The Actor Model 22
3.2.2 Ptolemy II 23
3.2.3 VisualSense 26

4 The Callas Programming Language **29**
4.1 Programming Language 29
4.1.1 A Callas Program for a Sensor Node 31
4.1.2 A Callas Program for a Sink Node 32

4.1.3 Callas Network Application 32
4.2 Virtual Machine 34

5 Simulation of Callas Applications **35**
5.1 The Callas Virtual Machine as a VisualSense Actor 35
5.2 Network and Node Simulation Models 37
5.2.1 Network Model 38
5.2.2 Node Model 38
5.3 Automatic Generation of Simulation Models 40
5.4 Performance and Scalability 41

6 Integrating WSN Simulation into Workflow Testing and Execution **43**
6.1 A Logistics Scenario 43
6.2 Workflow Execution Integrating WSN Simulation 46
6.2.1 Integrating WSN Simulation in Kepler 47
6.2.2 Workflow Interoperability 48

7 Conclusion **51**

Bibliography **62**

List of Figures

2.1 Generic sensor node hardware architecture 8

3.1 A counter program written in Simple Actor Language 22
3.2 A Ptolemy II model with atomic actors 24
3.3 A Ptolemy II model with a composite actor 25
3.4 A VisualSense network analysis model 26
3.5 A VisualSense sensor node model 27

4.1 Callas type declarations 30
4.2 A Callas type declaration that serves as a network interface 30
4.3 A Callas program for a sensing node 31
4.4 A Callas program for a sink node 33
4.5 A Callas network program 33

5.1 Configuration of VisualSenseVM and of the adapted CVM 36
5.2 A VisualSense model of a network containing five nodes 38
5.3 A generic VisualSense model of a node 38
5.4 A Callas network file extended with generator parameters 41
5.5 Simulation duration and memory footprint 42

6.1 Callas types for a logistics application 44
6.2 Interface type for a logistics application 44
6.3 Sensing node program for a logistics application 45
6.4 Sink node program for a logistics application 45
6.5 Callas application for a logistics scenario 46
6.6 Simulation model to be integrated in Kepler 47
6.7 Kepler workflow where TruckNetwork encapsulates the WSN model 48

List of Tables

2.1 Characteristics and prices (as of 2010) of five sensor devices 8
2.2 Strong points of five sensor devices 9
2.3 Network concerns and protocols for WSN, grouped by network layer . . . 11
2.4 Main standards and technologies for WSN, grouped by network layer . . 12
2.5 A classification of WSN programming languages 16

3.1 Main concerns in wireless sensor network simulators 20
3.2 A characterisation of Open Source, generic WSN simulators 21
3.3 Project information of Open Source, generic WSN simulators 21

5.1 Measurings of simulation duration and memory footprint 42

Chapter 1

Introduction

A sensor network is a collection of devices that can collectively measure some scalar or vector field and communicate the resulting data to a *base station*, for instance, or to an *actuator* that can perform some action on the environment. Wireless Sensor Networks (WSN) are a special case of sensor networks where the communication is, as the name implies, wireless, the number of nodes is usually large, and the nodes have very limited computational power and energy autonomy.

WSN are a promising field in terms of practical implications of technology on our everyday life. Current WSN applications include environmental monitoring, extreme precision farming, health care, warfare, security, and logistics, to name a few. Envisioned applications include biomedical research and space exploration.

The topic of WSN has attracted the attention of both companies and research groups. The challenges raised in terms of hardware, such as device miniaturisation, or energy autonomy improvement, are as important as those raised at the software level, particularly in what concerns the operating systems and the programming languages for these devices. In spite of the advances in both areas [4, 49, 85], programming is still seen as the weakest link in WSN [55].

A typical WSN would have nodes running nesC [28] code on TinyOS [32], an event-driven operating system. This approach is somewhat low level and has disadvantages in terms of network reconfiguration, for instance. While there have been some attempts to provide an abstraction level on top of TinyOS, most are not formal-based, which makes it impossible to prove their correctness. A notable exception is Callas [54], a (type-)safe programming language for WSN that may serve as an intermediate language upon which high-level, type-safe programming abstractions may be encoded.

Simulation is another active research area in WSN. The potentially large number of nodes compromises the *testing* of applications for WSN, moreover, WSN are often deployed at remote locations, making physical access to the devices difficult, or even impossible. For such applications, simulation may play a decisive role in what concerns testing and debugging [22]. WSN simulators are computer systems that run WSN applications in

virtual environments where geographical properties, radio communication, and physical phenomena are modeled. There are several WSN simulators, with different characteristics. In this work, VisualSense [7], an Open Source, generic (not specific to a given node architecture) simulator is used.

Creating simulation models is a laborious task. It is necessary to define the sensor nodes (or at least the sensor nodes properties) and to specify their positions, the WSN application, the physical environment, the radio properties, and other aspects. Ideally, one would be able to automatically generate simulation models from WSN applications and a set of simulation parameters.

This thesis explores automatic simulation model generation. In particular, an approach for generating VisualSense models from Callas applications is presented. We present a generator tool that creates simulation models, for the VisualSense simulator, from Callas applications. We have presented this tool in [72] In addition, we present a means of integrating VisualSense models in a workflow management system. This integration, that we have presented in [73, 74] aims at easing the simulation of business processes in areas where WSN have applications, such as logistics.

Motivation

Sensor networks are gaining momentum in various fields, notably in industrial and environmental monitoring, and more recently in health care, logistics, and other areas. Being a relatively novel subject, WSN are under active research, whether in terms of hardware, communication protocols, operating systems, and programming languages.

Wireless sensor networks are hard to program, deploy, and test. Testing WSN is hard because the networks are usually large and can be deployed in wide areas, or in harsh environments. WSN simulators are often used to test the applications prior to deployment. A simulator serves as a sandbox where it is possible to control virtually all aspects of a WSN, namely, the physical environment, nodes, application, and physical phenomena.

This work addresses a key problem of testing WSN using a simulator: simulation model definition, a laborious (manual) task. The general goal is to automatically generate simulation models directly from WSN applications and a set of simulation parameters, thus easing WSN testing and deployment. Furthermore, this thesis addresses testing higher level applications based on information from WSN, a topic that can only grow in importance as the number of applications for WSN continues to rise.

Goals

We explore an approach towards the goal of achieving simulation model generation. It consists of *i)* adapting the Callas Virtual Machine in VisualSense, adapting its interface

in a simulator component, *ii)* defining generic network and sensor model templates that serve as building blocks for Callas network models, and *iii)* creating a simulation model generator tool for this approach. Moreover, this thesis explores the possibilities of using WSN simulation at higher (application) levels and presents one of such possibilities: integrating Visualsense's sensor network simulations directly into the execution of workflows in the Kepler workflow management system.

Structure of the document

The structure of this thesis is as follows. Chapter 2 introduces the subject of WSN, giving some insight into WSN applications, devices, communication, and programming models. Chapter 3 explores WSN simulation, presents a comparison of WSN simulators, and then details the VisualSense simulator, used through the rest of the thesis. Chapter 4 presents the Callas WSN language, along with examples, and the Callas Virtual Machine, that serves as a run-time system for Callas. In Chapter 5, we pave the way for automatic simulation model generator and present the generator tool. In Chapter 6, we integrate WSN simulation model in workflow execution and testing. Finally, Chapter 7 concludes the thesis and outlines future work.

Chapter 2

Wireless Sensor Networks

A Wireless Sensor Network is a potentially large collection of tiny devices that have physical sensing capabilities, and communicate wirelessly. In a WSN there are special nodes that leverage on the information gathered by the sensing nodes, namely, *actuators* and *sinks* (or *base stations*). Actuators perform some kind of action on the environment, based on the information received from the sensors; for instance, an actuator may trigger the cooling process of a nuclear reactor when a given temperature is reached. Sinks are usually nodes with much greater computational power than the sensors that usually serve as sensor network's base stations. A sink may log the information harnessed by the sensors, or even (re)configure the network.

WSN can be used for many purposes, ranging from industrial monitoring to biomedical research [4]. However, their widespread usage is still restrained by difficulties such as the very limited energy autonomy and computational capabilities of the sensor nodes, the the need for specialised communication protocols. In addition, testing and debugging networks composed of large numbers of nodes, or networks deployed in harsh environments, is hard, if not impossible.

In this chapter, we present an overview of WSN. Section 2.1 presents some of the current and envisioned WSN applications. Section 2.2 summarises the state of the art in sensor devices. Sections 2.3 and 2.4 give some insight into the communication aspects and into the programming models. Finally, in Section 2.5, presents the main research topics in the field.

2.1 Applications

The diversity of WSN applications stems from the variety of physical phenomena that can be sensed: temperature, sound, wind speed, magnetic fields, acceleration, light and non visible radiation, and concentration of substances in a given medium, to name a few [4]. WSN sensing capabilities are used for environmental monitoring, extreme precision farming, security, warfare, and scientific research. More modest, but nonetheless

commercially relevant applications, include heating and ventilating [4]. In spite of the aforementioned diversity, WSN applications can be classified into three categories: monitoring (*e.g.*, temperature monitoring), tracking (*e.g.*, vehicle tracking) and research (*e.g.*, space exploration). In the remainder of this Section, some current and envisioned applications are presented.

2.1.1 Current Applications

Current applications usually fall under the monitoring and tracking categories. The following project (and product) examples represent a very small fraction of what is already done with WSN.

Forest Fire Detection SISVIA [18], developed by the spanish company dimap [17], is the first real-world WSN-based forest fire detection system. It was deployed in a northern region of Spain, in 2009, and covers 2 km^2 with 90 waspmote [46] sensors that monitor temperature, relative humidity, carbon monoxide, and carbon dioxide every 5 minutes. These parameters are communicated to a control centre by 2 special nodes that act as gateways. Each node is connected to a solar panel that recharges the battery, making the WSN autonomous in terms of energy.

Ocean Monitoring The ARGO [62] project provides ocean data that is being used to understand the ocean currents. Its network consists of 3,000 nodes, distributed in all the oceans, that can dive to a depth of 2,000 meters, and then emerge and transmit the collected data (pressure, temperature, and salinity) to a satellite.

Glacier Monitoring PermaSense [66] is a geo-monitoring system that provides data about the permafrost at the Swiss Alps, allowing to perform hazard assessment to tourist resorts and other man-made infrastructures. The network remains unattended for most of the year because of the extreme weather conditions. The nodes endure temperatures as low as -30^{o} C.

Wildlife Monitoring In Kenya, at the Mpala Research CenterKenya, the ZebraNet [62] project takes advantage of a WSN to study the behaviour of wild horse, zebra, and lion populations. The sensors used in the animals are able to receive GPS information and to measure the ambient light, allowing to estimate the movement patterns of single animals and groups, as well as interactions between species. Whenever two sensors are in range, they exchange their data. On a regular basis, a mobile sink reads the data from the sensors in range.

Health Monitoring Sleep Safe [84] is a monitoring system that can help prevent Sudden Infant Death Syndrome. A sensor monitors the infant's sleeping position and alerts the parents when the infant is lying in its stomach.

Industrial Monitoring Soflinx Corporation, a security sensor network supplier, provides a perimeter security system that enables real-time detection of several hazardous substances in industrial environments. The system includes actuators that react automatically based on the collected data [4].

Mining Monitoring Mining is a dangerous activity in great part because even slight structural changes in tunnels may cause collapses. SASA [45] is a WSN based monitoring system that can detect, locate, and report collapse holes. The system can automatically detect and reconfigure nodes displaced by a collapse.

Military Tracking There are several WSN systems for tracking military ground vehicles. The nodes can be deployed from unmanned aircrafts and are able to cooperatively estimate the path of the vehicles and then transmit the results to another unmanned aircraft that flies by in order to collect the data [62].

2.1.2 Envisioned Applications

The future holds many applications for WSN. As the following examples demonstrate, the key factors holding back envisioned applications are the sensor node's size and energy autonomy.

Health Monitoring Implanted wireless biomedical sensors in diabetic patients could be used to assess the glucose levels in the blood. The device could also be an actuator that injected insulin as needed [4].

Biomedical Research Some envisioned applications require much smaller sensors to be developed. This is especially true in nanomedicine [3], where sensors smaller than a cell would provide data for the analysis of cell functions at the molecular level.

Space Exploration WSN could act as a distributed probe, making measurements in (or around) celestial bodies. WSN could also replace much of the wiring in spacecrafts [20].

2.2 Sensor Devices

A basic sensor device is composed by four components: processor, transceiver, power source, and sensing unit [4]. More complex devices may have external memory, or ac-

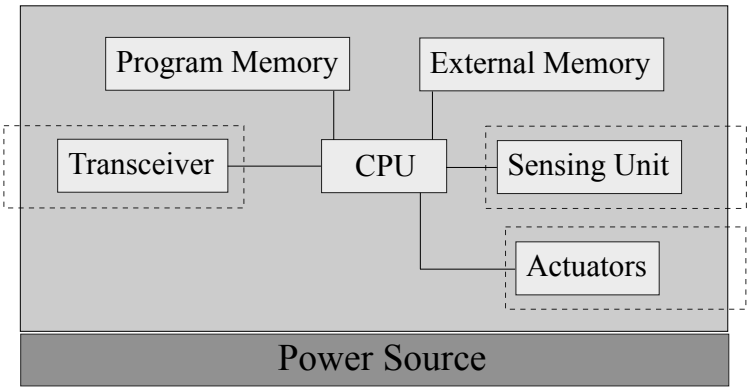

Figure 2.1: Generic sensor node hardware architecture

Table 2.1: Characteristics and prices (as of 2010) of five sensor devices

Sensor	**C.P.U.**	**Program Mem.**	**External Mem.**	**Programming**	**Max. Auton.**	**Unit Price** (Eur)
Particles uPart	Microchip 12F675	64 bytes	1.75 KB Flash	In circuit	6 years	15
WeBee3	Intel 8051	8 KB	128 KB Flash	C	10 years	10
Waspmote	Atmega 128L	8 KB	128 KB Flash 2GB SD Card	C Processing	1 year	99
SunSpot	ARM 920T	512 KB	4 MB Flash	Java (Squawk JVM)	1 day	210
IMote 2.0	ARM 11	32 MB	32 MB Flash	NesC C# (.Net Micro)	NA	NA

tuators [55]. Figure 2.1 illustrates a generic architecture of a sensor device. The dotted lines around some components indicate that they can be replaced. This is not true for the smallest, cheapest node, but is common in the more larger, configurable nodes.

The main concerns in sensor devices are energy autonomy, computational power, size, and cost. These are, naturally, intertangled. Although there have been improvements on sensor hardware and miniaturisation, there is still a long path to run [4, 22]. For example, a device with an increased computational power is tipically less (energy) autonomous.

Many sensor devices have been developed, both academically, and commercially. Five devices representing the current sensor market are described in Table 2.1, in terms of Central Processing Unit, program memory, external memory, programming model/language, maximum energy autonomy, and unit price (as of 2010). In almost every parameter there is a very significative variance; program memory, for instance, ranges from 64 bytes to 32 MBytes. The table also depicts the correlation between computational power and energy autonomy: nodes with higher computational power deplete their batteries much faster. Table 2.2 summarises the strong points of the considered sensor devices.

Table 2.2: Strong points of five sensor devices

Sensor	Evaluation
Particles uPart	Very small size (less than 1 cm^3, battery included).
WeBee3	Very low price; Extended battery lifetime.
Waspmote	A balanced choice between computational power, battery lifetime, and price; Modular architecture, allowing to add/remove sensor and communication boards; Plenty of accessories, including a solar panel that turns the node virtually autonomous; Commercial support.
SunSpot	Computational power.
IMote 2.0	Computational power.

Since there is yet no glimpse of an ideal sensor, it is not possible to dissociate the sensor device from the WSN purpose and, therefore, must choose devices by selecting the desired features at the expense of others.

2.2.1 Nanosensors

Nanotechnology is to become an enabling technology in sensor development. Once accomplished, nanosensors (sensors in the nanometer scale) will make use of the unique properties of nanomaterials and nanoparticles to detect and measure events in the nanoscale. Nanoactuators, like "normal" actuators, will perform some kind of action on the environment, based on the data provided by the nanosensor nodes. Depending on the nature of the sensing capabilities, nanosensors and nanoactuators can be classified as physical, chemical, or biological [53].

Wireless Nano Sensor Networks (WNSNs) can have environmental, industrial, military, and biomedical applications. In particular, WNSN can have a large impact in health monitoring systems: sodium, glucose, cholesterol, cancer biomarkers, and other substances may be monitored in blood by means of nanosensors. For instance, nanosensors could monitor the glucose level in blood and transmit the data wirelessly to a cellphone wich, in turn, could forward the data to a healthcare provider [3].

In recent years, many advances have been accomplished in nanotechnology. In Berkeley, a nanoradio of about ten nanometers in diameter (and several hundred nanometers long) was built and used to perform a FM broadcast across a room [38]. In Harvard, a method for assembling and disassembling nanowires was created [83]. In spite, many more advances must be made before nanosensors can become a reality [3].

As with normal sensors, energy autonomy is a key issue in nanosensors. Although nanomaterials could be used to manufacture nanobatteries with high power density, every

battery needs to be recharged. The concept of self-powered, nano-devices has been recently introduced as a solution to overcome the energy autonomy problem. Nanosensors could harvest mechanical (*e.g.*, from human body movements), vibrational (*e.g.*, from acoustic waves), or hydraulic (*e.g.*, blood flow) energy from the environment and convert it into electric energy [3].

2.3 Communication

A typical WSN is *a*) a wireless ad-hoc network, meaning that there is no preexisting infrastructure, and *b*) a multi-hop network, meaning that several nodes may forward a given message from a node to a sink. WSN communication can be characterised in terms of the path in which it takes place: forward and reverse. The former concerns the flow from the sensors to the sink, while the latter concerns the flow from the sink to the sensors. Generally speaking, the reverse path requires higher reliability, because messages may contain code to be deployed, for instance. However, the forward path may also require high reliability, specially if some events in the environment must be detected and forwarded to the sink [4].

Transmitting is, arguably, what drains more power from the node's battery, and, therefore, what influences most the network lifetime. The approach used to characterise the network lifetime may be based on the first node to wear out its battery, or some other more complex criteria that analyses the network connectivity, regardless of individual nodes with depleted batteries. Power preservation is vital to extend the network lifetime, hence communication protocols must be very efficient [8].

2.3.1 Protocols

WSN require communication protocols that take into account power conservation and the potentially large number of nodes. Furthermore, the choice, or development, of a protocol is determined by the purpose of the network, or class of networks. Table 2.3 presents the main communication concerns and protocols for WSN, grouped by layer. Further details on the protocols therein can be found in [86].

Not all WSN protocols follow the (traditional) layered approach. Cross-layered protocols minimise overhead and can be more energy efficient [84] than their counterparts. In this kind of protocol, the layer interaction varies: SP unifies the Network and Data Link layers, while JOCP unifies all the Transport, Network, Data Link, and Physical layers.

Security Networks can be subject of passive attacks, where there is no traffic modification (*e.g.*, eavesdropping) and active attacks, such as denial-of-service (inhibition of communication), masquerading (access to resources by an attacker pretending to be an authorised user), replay (retransmission of stored messages), and message modification

Table 2.3: Network concerns and protocols for WSN, grouped by network layer

Layer	**Concerns**	**Protocols**
Transport	Congestion	STCP
	Reliability	PORT
	Energy conservation	GARUDA
		CODA
		DST
		PSFQ
		ESRT
Network	Routing	Geographical Routing
	Scalability	ALS
	Synchronisation	SecRout
	Data cache	SCR
	Data aggregation	
	Computation overhead	
	Communication overhead	
	Data security	
	Energy requirements	
Data Link	Channel access mode	TRAMA
	Time synchronisation	B-MAC
	Protocol type (TDMA, CSMA, CA)	Z-MAC
	Energy conservation	Low Power reservation based
		Low Power distributed MAC
		CC-MAC
Physical	Bandwith choice	
	Radio	
	Modelation scheme	

Table 2.4: Main standards and technologies for WSN, grouped by network layer

Layers	Standards	Technologies
Transport Network Upper layer of Data-Link	ZigBee [87] WirelessHART [39] Wibree [82]	
Lower level of Data-Link Physical	IEEE 802.15.1 [35] IEEE 802.15.3 [36] IEEE 802.15.4 [37]	

(new, changed, or re-ordered messages are sent to the network). Some protocols address the security issue: SPINS (Security Protocols for Sensor Networks), for instance, ensures data confidentiality, two-party data authentication, data freshness, and authenticated broadcast.

2.3.2 Standards and Technologies

Several standards have been proposed for WSN communication. As with protocols, the choice, or development, of a standard is determined by the purpose of the network, or class of networks. Table 2.4 presents some of the WSN standards, technologies, and their relation to the protocol stack.

IEEE 802.15.1 Specifies the physical layer and the medium access control (MAC, lower level of the data-link layer) for Bluetooth communication, where the radio operates at 2.4 GHz.

IEEE 802.15.3 This standard targets real-time, multi-media streaming. It specifies the physical layer and MAC for high data rate for Wireless Personal Area Networks (WPAN). The physical layer operates on a 2.4 GHz radio and supports data rates from 11 to 55 Mbps.

IEEE 802.15.4 Specifies the physical layer and the MAC for low data rate WPAN. The physical layer supports the 868/915 MHz low bands and the 2.4 GHz high band, and MAC uses the CSMA-CA protocol. This standard addresses wireless sensor applications that require short range communication, low energy requirements, and low cost.

ZigBee Builds on IEEE 802.15.4, defining the higher layers of the protocol stack. It is intended to enable networks containing thousands of low cost, low power devices. ZigBee devices are divided into three types: *i*) coordinators, nodes that initiate network formation and can bridge networks together, *ii*) routers, that allow multi-hop communication, and

iii) end-devices, sensor or actuator nodes that communicate solely with the routers and coordinators.

WirelessHART This standard targets process measurement and control applications. Like ZigBee, it builds on IEE 802.15.4.

Wibree This technology builds on the IEE 802.15.1 standard, providing low cost and low energy communication for Bluetooth devices.

DASH7 This technology implements the ISO/IEC 18000-7 standard for RFID. By operating on the 433 MHz frequency, DASH7 devices have a range of more than 1 Km, use less power, and can transmit through concrete and water. DASH7 does not support streaming, nor synchronisation.

2.3.3 Gathering

Gathering is the process of transmitting data from the sensor nodes to the sink, possibly over multiple hops. In order to achieve extended network lifetime and scalability, this process must be efficient. Clustering, aggregation, and inference are techniques that improve gathering efficiency by reducing the number of messages on the network.

Clustering protocols, such as LEACH, allow networks to dynamically form clusters in which a head node is responsible for the communication with neighbour clusters). Aggregation protocols use several approaches to limit the number of nodes that a given node is allowed to communicate with. For instance, using PEDAP, the sink node computes routing tables based on the location of the nodes. Whenever a node fails, a new routing table must be computed.

For many applications not all the collected data set is useful. In fact, for applications like temperature monitoring, only a small set of values is necessary, *e.g.*, the maximum and minimum temperatures. Inference is a distributed computing technique that can help reduce the network traffic. For instance, a node could only forward a message if it has a bigger value that the last maximum received, or smaller than the last minimum received.

2.4 Programming Wireless Sensor Networks

Programming WSN is difficult because applications these type of networks are (large-scale) distributed programs that must run on devices that are very limited in terms of hardware and also in terms of energy autonomy [55]. Presently, most sensor nodes run module-based operating systems (*e.g.*, TinyOS [32]) and are programmed in nesC [28] or TinyScript/Maté [42]. There are important limitations in this low-level approach [8], namely:

- Lack of a global vision of a sensor network application;
- Absence of a dynamic means of network reprogramming, resulting in the necessity of individual sensor reprogramming, which is unfeasible for large WSN, where massive code deployment is desirable;
- Absence of a rigorous model of the sensor network at the programming level, which would allow for formal verification of program correctness.

Although these limitations are well known and there has been a number of proposals that aim to surpass them, very few WSN rely on higher-level programming models [55].

2.4.1 Operating Systems

A typical WSN operating system provides few abstractions; it supports a programming language, and a low level communication facility. In order to reduce overhead and memory usage, usually only a selection of the operating system modules are deployed to sensor nodes, according to the application needs.

Most current sensor networks run on top of the TinyOS [32] operating system and its sibling programming language nesC [28]. TinyOS provides a very simple, event-based, single-threaded execution-model with non-preemptive tasks. The system is loaded onto the sensor nodes as a set of modules to be used by a target application.

Other operating systems for WSN have been proposed. Contiki [21] is also event-driven, but, unlike TinyOS, supports multi-threaded execution, and dynamic loading of program modules. MANTIS [10], Nano-RK [24], and BTnut [9] operating systems support preemptive multi-threading, meaning that the operating system, not the application programmer, manages the CPU. As a side note, Nano-RK provides control to hardware resources in order to support real-time WSN applications.

There are also systems, such as the Squawk JVM [65] that run directly on the hardware, without operating system support. The Squawk JVM supports preemptive multi-threading.

2.4.2 Programming Models and Languages

Traditionally, WSN have been programmed at a very low-level of abstraction, however, there is a growing trend towards the development of high-level programming models and abstractions for these networks. A programming model reflects a given point of view. In the following we present four programming models common in WSN:

- **Stream.** The programmer sees the network as a data stream, with no perception of the underlying hardware;

- **Region.** The network can be partitioned into groups of sensors, according to some membership criteria, and programmed on a partition base;
- **Database.** The network is seen as a dynamic data repository that may be queried by declarative languages such as SQL;
- **Computing.** The programmer perceives the network as a distributed system that may perform online computation, by hosting autonomous mobile agents, for instance.

A programming model can be implemented by a programming language in various ways, therefore, it is not sufficient to classify a language. In order to be able to classify the main WSN programming languages, we follow the criteria on [48]. It is based on three items: hardware interaction, network perception, and data acquisition, summarised below. Table 2.5 lists several languages classified according to this criteria.

Hardware Interaction *i)* **Low-level**, system-level programming of the sensor networks, where programs make direct calls to the operating system, *ii)* **Virtual Machine**, programs run on a software infra-structure that creates an abstraction layer over sensor specific hardware and operating system, while allowing the programmer to retain some fine grained control of the applications, *iii)* **Middleware**, programming using an API, provided by an underlying middleware, that hides the details of the sensor network from the programmer, and *iv)* **High-level**, programming with very high-level abstractions of the network that hide all networking and communication details. Programs are distributed applications that are generally not targeted at a specific sensor network architecture or configuration.

Network Perception *i)* **Macroprogramming**, WSN applications are developed as typical distributed applications, without requiring the developer to specify the behaviour of each computing node individually. The low-level details of communication and network architecture are abstracted away, and *ii)* **Sensor-based**, WSN applications are developed with the network architecture and, in some cases, with node hardware details in mind.

Data Acquisition *i)* **Communication-centric**, lowest level data abstraction where data in the network is seen as messages; *ii)* **Data-centric**, high-level data abstractions, namely streams and databases; and *iv)* **Computation-centric**, mobile agents evolve to allow communication and in-network re-programming.

Generally, high-level languages such as TinyDB [52] are compiled into low-level languages, in the case, nesC [28]. It is rather difficult to ensure that the semantics of the high-level application is equivalent to the semantics of its low-level version. Moreover,

Table 2.5: A classification of WSN programming languages

Language	Hardware Interaction	Network Perception	Data Acquisition
Abstract Regions [51]	Low-level	Macroprogramming	Message
Agilla [26]	High-level	Sensor-based	Mobile Agent
Cougar [27]	High-level	Macroprogramming	Database
Deluge [33]	Low-level	Sensor-based	Message
DTM [64]	Virtual Machine	Macroprogramming	Message
EnviroTrack [1]	Middleware	Sensor-based	Message
HOOD [81]	Low-level	Macroprogramming	Message
Impala [47]	Middleware	Sensor-based	Message
IrisNet [59]	High-level	Macroprogramming	Database
Kairos [30]	High-level	Macroprogramming	Stream
Maté [42]	Virtual Machine	Sensor-based	Message
Mottle [43]	High-level	Sensor-based	Message
nesC [28]	Low-level	Sensor-based	Message
Regiment [56]	High-level	Macroprogramming	Stream
SensorWare [11]	High-level	Sensor-based	Mobile Agent
SpacialViews [57]	High-level	Sensor-based	Mobile Agent
Squawk [65]	Virtual Machine	Sensor-based	Message
TinyDB [52]	High-level	Macroprogramming	Database

the low-level programming languages lack a formal specification, making it difficult to reason about program properties.

This situation emerges from the absence of an adequate hardware abstraction, *e.g.*, a virtual machine, that allows a complete formal specification for the semantics of sensor network applications. Some important work has been made in this direction, most notably Maté and the Squawk JVM. The problem of lack of specification at the language level is tackled by Callas, a formal-based, type-safe language for WSN that we use along this thesis.

2.5 Research Topics

WSN are a relatively novel subject. While they have drawn a lot of attention in recent years, the constraints imposed by current hardware, communication protocols, and energy autonomy, for example, lead to believe that the subject is to remain under active research.

Below, we highlight some research topics. Some concerns, such as energy autonomy, or cost, are transversal to most of the research topics.

Sensor Devices Current sensor devices are very limited. An ideal sensor would be cheap, small, have enough battery to allow its operation for many years, or be able to harvest energy from the environment, have a reasonable computational power, in order to allow the deployment of higher level applications, and would be resilient, in order to allow

large scale deployments from aeroplanes, boats, and other vehicles. Furthermore, for many envisioned applications, sensors must be developed in the micro and nano scale [3, 4].

Communication Protocols and Security Communication protocols must have better performance and energy efficiency, while providing Quality-of-Service. Protocols must also be secure, ensuring that all the stack layers are safe from malicious attacks (current secure protocols address mainly the data-link and network layers) [84].

Programming Abstractions and Correctness Most programming languages for WSN provide few abstractions, whether in terms of language constructs, or in terms of communication [55]. Furthermore, the abstractions are typically not formal based, hence, their correctness is not guaranteed [54].

Fault Tolerance Faults in WSN can be originated by hardware, nodes running out of power, and temporary erroneous readings (transient faults). Currently, the exclusion of nodes with depleted batteries does not obey to time bounds, for example, and there is little support for the detection of erroneous readings [55].

Node Mobility Mobile nodes and sinks have special requirements, such as dynamic network topology and delay tolerance, that are not expressed in programming abstractions. Therefore, neighbour discovery and other concerns are dealt with in a per-application basis [55].

Debugging and Testing Debugging and testing WSN is carried out using testbeds and simulators. There is currently no support for debugging and testing at the programming level [55].

Chapter 3

Wireless Sensor Network Simulation

The deployment of WSN poses great challenges. At the node level, the communicational and the computational capabilities are constrained by severe energy and hardware limitations. At the network level, the potentially large number of nodes implies that it may be onerous or even unfeasible to test an already deployed network. WSN simulators are computer systems that allow us to deploy and test networks in virtual environments that serve as sandboxes to experiment with communication protocols, radio signal properties, and physical sensing. Therefore, by providing a way of testing and debugging WSN applications in (simulated) environments, simulation can play a decisive role in WSN during testing and deployment.

Another approach towards WSN testing consists of using *physical testbeds*, such as GNOMES [80], or S-Net [13]. Physical testbeds consist of a set of devices to which sensor nodes are connected in order to be monitored. The gathered data can then be used to infer the network behaviour. Naturally, physical testbeds are not suited for large network testing, because a lot of effort (and hardware) would be required. Some authors note that there has been improvements in mathematical analysis and experimental deployments but, however, simulation is still the preferred tool for the study of WSN [22].

In this Chapter, section 3.1 introduces the topic of WSN simulation and draws a comparison of popular WSN simulators. Section 3.2 presents the VisualSense WSN simulator, giving some insight into the simulator internals and model definition.

3.1 Wireless Sensor Network Simulators

WSN simulators are complex programs. A simulator should handle large numbers of nodes without degrading performance significantly, model the radio channel, battery discharge, physical environment, and support different communication protocols [22]. Ideally, it should also provide a graphical interface. Table 3.1 presents the main concerns regarding WSN simulation, grouped by communication protocol support, environment modelling, and graphical support.

Table 3.1: Main concerns in wireless sensor network simulators

Protocol support		**Environment Modelling**		**Graphical Support**	
C	Classical (*e.g.*, TCP/IP, Ethernet)	B	Battery	E	Edition
A	Ad-hoc (*e.g.*, MANET protocols, AODV, DSR)	PP	Physical Phenomena	A	Animation
W	Wireless (*e.g.*, propagation, mobility, IEEE802.11)	PE	Physical Environment	D	Debugging
WSN	Some common WSN Protocols (*e.g.*, Directed Diffusion, S-MAC)				

WSN simulators are typically based on discrete event simulation. In this type of simulation, the operation of a system is represented as a chronological sequence of (discrete) events. An event occurs at a given instant in time and sets a new state in the system [1]. The new state may trigger another event, and so on. Discrete event simulation requires a global clock and an event queue to manage the chronological sequence of events. Although discrete event simulation can model many systems, it is naturally not suited for every class of system. For instance, physical phenomena and battery discharge requires continuous time simulation. Modelling systems that have discrete and continuous time components is far from trivial.

WSN simulators can be divided into two categories: architecture specific, that model WSN for a given node architecture (*e.g.*, TOSSIM [44], for the Berkeley Motes), and general simulators, that allow user-defined sensor nodes (*e.g.*VisualSense [7]). Architecture specific simulators include ATEMU [61], EMStar [29], TOSSIM [44], Avrora [68]. These are not subject of study, since this work aims at a device-independent view of WSN simulation. Non specific but also non Open Source simulators include QualNet [77], and OPNET [25]. These are also not covered here. There are a number of general, Open Source simulators, however, most of them are either abandoned projects (such as JSim [75], SensorSim [60], and Sidh [14]) or do not provide important features such as an advanced GUI and environment modelling. Table 3.2 presents a summary of the features and characteristics of general, Open Source simulators, is presented. The characterisation is made against the concerns depicted in Table 3.1.

When choosing a WSN simulator, it is also necessary to assess the project status. In fact, many WSN simulators have been developed, but few remain active projects. Another important aspect is that node definition is usually done programmatically, thus, the programming language of the node definition API may also be critical. Table 3.3 presents the programming language and project status of the simulators in Table 3.2.

Table 3.2: A characterisation of Open Source, generic WSN simulators

Simulator	**Protocol Support**	**Environment Modelling**	**Visualisation**	**Additional Notes**
JiST/SWANS	W/A	-	A (Javis/NAM)	
OMNET++	C/W/A	-	E(lim)/A/D	
SENS	W	B/PE	-	
SENSE	W/A/WSN	B	A	
GloMoSim	C/W/A/WSN	-	-	Gave origin to a commercial WSN simulator, QualNet.
ns-2	C/W/A/WSN	-	A/D (using nam)	
SSFNet	C/W/A/WSN	-	Proprietary	
NCTUns	C/W/A/WSN	-	E/A/D	
PAWiS	C/W/A/WSN	B	E(lim)/A/D	Extends OMNET++
Castalia	C/W/A/WSN	B/PP/PE	E(lim)/A/D	Extends OMNET++
J-Sim	C/W/A/WSN	B/PP/PE	E/A/D	
VisualSense	C/W/A/WSN	B/PP/PE	E/A/D	VisualSense is part of the Ptolemy II project.

Table 3.3: Project information of Open Source, generic WSN simulators

	Language	**Latest Release**	**Active**
JiST/SWANS	Java/Jython	1.0.6 (03/2005)	Yes, under development.
OMNET++	C++/NED	4.1 (06/2010)	Yes, under development.
SENS	C++	- (01/2005)	No.
SENSE	C++	3.0.3 (04/2008)	Yes, maintenance only.
GloMoSim	Parsec	2.0 (12/2000)	No.
ns-2	C++/OTcl	2.34 (06/2009)	Yes. A newer version, ns-3, is under development.
SSFNet	C++/DML/Java	2.0 (01/2004)	No.
NCTUns	C	6.0 (01/2010)	Yes, under development.
PAWiS	C++/NED	2.0 (07/2008)	No.
Castalia	C++/NED	3.0 (08/2010)	Yes, under development.
J-Sim	Java/Jacl	1.3 (02/2004)	No. A patch was released in 07/2006.
VisualSense	Java	7.01 (04/2008)	Yes, under development.

```
def Counter(value)
  case operation of
    get: (client)
      send value to client
    increment: ()
      become new Counter(value + 1)
  end case

let c = new Counter(0) in
  send increment to c
```

Figure 3.1: A counter program written in Simple Actor Language

3.2 The VisualSense Simulator

VisualSense is a modelling and simulation framework for WSN that extends the (actor oriented) Ptolemy II framework [23]. Here, the actor model is briefly presented, focusing in the aspects relevant for the understanding of VisualSense. Then, the simulator is addressed.

3.2.1 The Actor Model

The actor model is a concurrent computational model proposed by Carl Hewitt [31], and later formalised by Gul Agha [2], where the computational primitives are called *actors*. An actor executes independently and interacts with other actors solely by asynchronous message passing, therefore, it does not share state.

An actor is defined by a mail address that identifies it, a mail queue that holds the incoming messages, and a *behaviour* that determines how to handle the messages in the queue and that only accepts a given set of message patterns, thus defining the actor interface. Messages that do not comply with the behaviour are ignored. The behaviour maps each accepted message pattern with an *operation*. While performing an operation, an actor can send messages and create new actors. When the operation is completed, the actor updates its behaviour, whether to a behaviour specified in the operation, or to the same one it previously had, if no replacement was specified in the operation. The newly specified behaviour will be used to handle the next message in the mail queue.

Within an actor system, or *configuration*, an actor can only send messages to its *acquaintances*: actors whose mail addresses are known to it. Acquaintances may be actors that a given actor *knows* from its creation (given as parameters), actors that it creates, or actors that it learns of while interacting with other actors. In the actor model, a computation is always triggered by an incoming message. This implies that there must be an external message to boot a given configuration. The actors whose mail address is know outside the configuration are said to be *receptionist* actors.

To better illustrate the actor model, Figure 3.1 presents a counter actor. It is written in

Simple Actor Language (SAL) [2], an actor model programming language. The Counter behaviour is defined with a parametric acquaintance, value, that is used to hold the counter state. The actor supports two operations: get and increment. The get operation is triggered by a message with two parameters: the name of the operation to perform, get, and client, the mail address of the requesting actor. The increment operation is triggered by a message with one parameter, precisely the name of the operation increment. The last two lines in the counter program are not part of the behaviour definition, they serve to create the counter actor, with zero as value, and increment it to one.

The get operation executes as one would expect from an asynchronous counter: a message with the current value is sent to the client. The increment operation shows an important aspect of the actor model: the counter *becomes* another counter, with an updated value, while maintaining its mail address. An actor must always replace its behaviour at the end of an operation, either explicitly, as in the increment operation, or implicitly, as in the get operation, where a **become self** command would be automatically issued.

Although an actor must replace its behaviour after processing a message, it can still maintain a state if the replacement behaviour is always the same and if acquaintances are used to hold the state between replacements. In this example, the acquaintance is a built-in value, while the counter would be an acquaintance of some other actor. The acquaintance relations can be used to achieve actor composition, in the sense that an actor uses other actors as its components.

In the actor model there is unbounded nondeterminism because the arrival order of the messages is arbitrary and the mail queues are unbounded. However, there is also fairness, since all messages are guaranteed to be delivered. This does not mean that a given message will be meaningful when it arrives, since the destination actor may have changed its behaviour in the meantime.

The support for the actor model in programming languages varies in nature. Some languages, such as Clara [19], might be called pure actor languages. Other languages, such as SALSA [71] and Stage [6], extend or modify existing languages (Java and Python, respectively). There are also languages that provide actor libraries, *e.g.*, Scala [58].

The actor model is frequently used in modelling and simulation frameworks. It enables composition (using other actors, acquaintances), and encapsulation (message passing is the only form of communication). Therefore, it eases reusability. Besides Ptolemy II, the Simulink [78], LabView [76], and VHDL [79] frameworks are based on the actor model as well.

3.2.2 Ptolemy II

Ptolemy II is an actor-oriented component assembly framework for modelling, simulation, and design of concurrent, real-time, and embedded systems [23]. In Ptolemy II, an actor configuration is called a model and is persisted in Modelling Markup Language

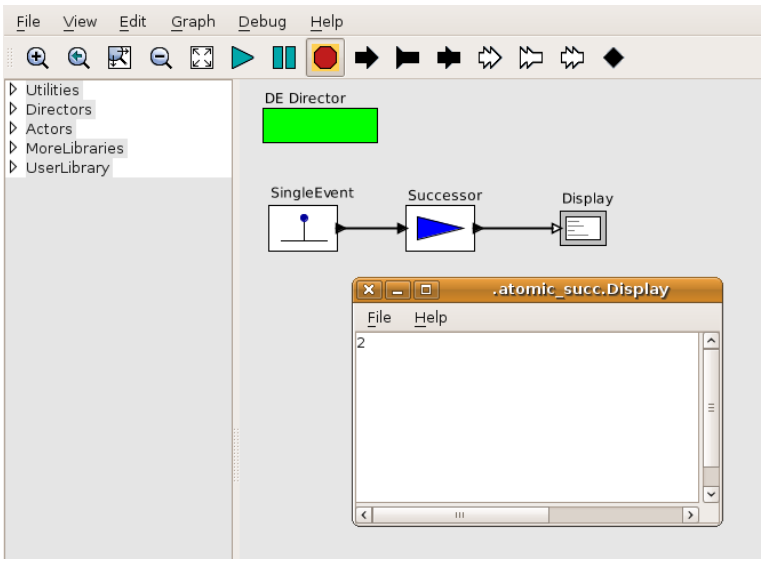

Figure 3.2: A Ptolemy II model with atomic actors

(MoML), an XML dialect. The framework has a GUI that allows drag-and-drop model definition, provides access to a library of Ptolemy II actors that perform operations ranging from signal analysis to matrix handling, and supports loading user-defined actors written in Java.

The two key concepts of the framework are *actors* and *domains*. Although communication is asynchronous in the actor model, synchronous interaction can be achieved as a special case of the more general asynchronous interaction. Communication can also be described in terms of being discrete or continuous in nature. In spite of executing independently and not sharing state, actors may share some aspects of their behaviours, particularly in what concerns communication. Ptolemy II captures these different patterns and concerns as domains.

A domain is implemented by a special component called *Director* that governs the communication between actors. For example, the Discrete Event domain is suitable to simulate digital circuits, where the communication is discrete, while the Continuous Time Model is appropriate to simulate analog circuits, where the communication has a continuous nature. The latter is also useful for modelling physical systems that can be described with differential equations.

In the "pure" actor model, the behaviour can be seen as an interface that defines the acceptable message patterns, whereas in Ptolemy II, the actor interface defines both the input and output message patterns, as *ports*. Also, actors do not interact directly, instead, the connections between them are mediated by *relations* (a sort of wires for actors) to which the actor ports are *linked*. In Ptolemy II, actor ports are typed in order to enable type safety. The type system ensures that a port only receives messages of its type, or subtype. Figure 3.2 presents a Ptolemy II (discrete event) model containing *i)* a Discrete Event

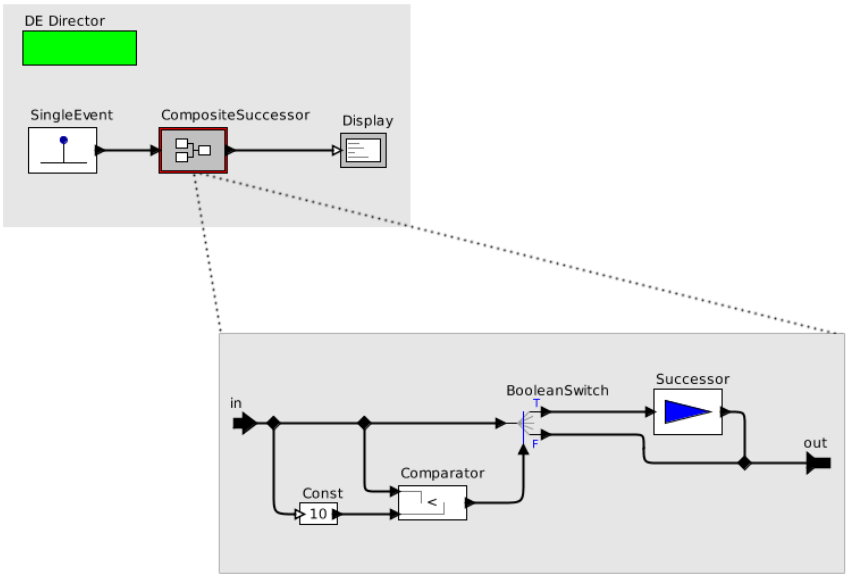

Figure 3.3: A Ptolemy II model with a composite actor

director, *ii)* a Single Event actor, configured to send the value "1" at the start of the simulation execution, *iii)* a Successor actor, that adds "1" to every value received, and *iv)* a Display actor, that prints the incoming messages in a GUI window.

A Ptolemy II actor may be atomic, or composite, where an actor contains other actors. In Figure 3.3, the Successor actor is replaced by the CompositeSuccessor, that limits the maximum output value to "10". The notion of actor composition in Ptolemy II is similar, although more restrictive, than the notion of acquaintance in the actor model. In fact, while in the pure actor model acquaintances are used both to connect actors and to compose actors, in Ptolemy II acquaintances are established among actors at the same level of hierarchy, but composition is accomplished by composing actors themselves.

Besides reusability and encapsulation, actor composition serves a very important purpose: domain heterogeneity. An actor may contain a model that implements another domain. This is useful, for example, to model a sensor that reads some physical value from the surrounding environment. The environment would be modelled using the Continuous Time domain and would be embedded in the Discrete Event where the sensor would be modelled. Ptolemy provides actors that take care of the interaction between the different domains. In more abstract terms, a model is a hierarchical graph where actors are the entities and relations are the arcs. An entity may be either atomic or composite. In the latter case, the entity defines a scope where a domain may be applied.

It is important to distinguish simulated time from simulation time. The former refers to the time interval that is simulated, for instance, ten minutes of a temperature sensing WSN, while the latter refers to the amount of time that it takes for the simulator to perform

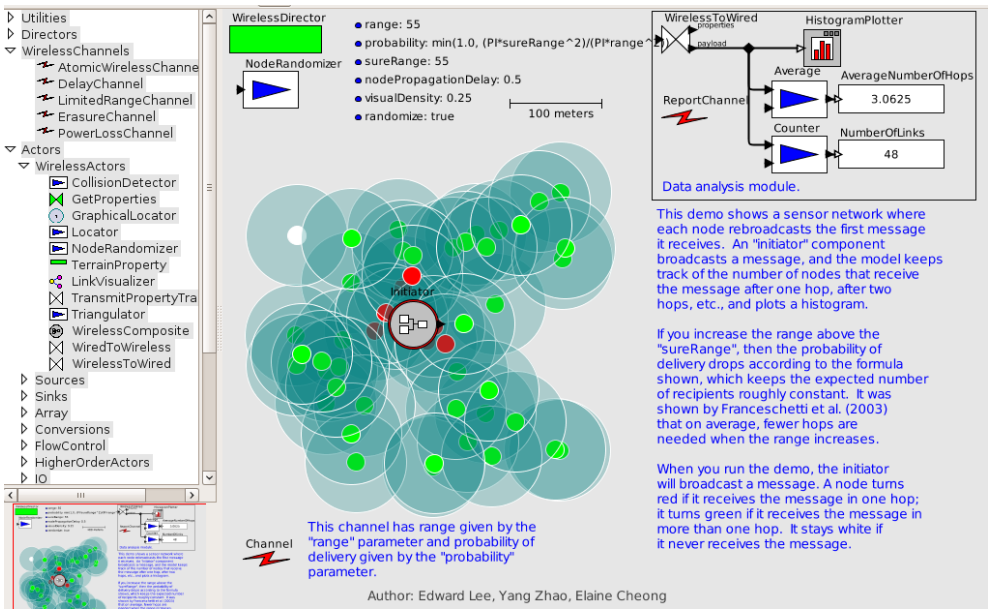

Figure 3.4: A VisualSense network analysis model

the simulation. It may be less than, equal to, or more than the simulated time, depending on the model complexity, the simulator settings, and the hardware platform.

In discrete event simulation, at each simulation clock tick, the events scheduled for that time are triggered (the clock may also "jump" to next queued event). The triggered events will eventually cause the scheduling of other events, simulating the model's (discrete event) behaviour. The Discrete Event director handles simultaneous events by deterministically adding an index i to the event time-stamp s_t, obtaining $s_{t,i}$. The simulation time is therefore two dimensional; the events e_a, e_b, e_c, with time-stamps $s_{1,1}$, $s_{1,2}$, $s_{1,3}$ all occur at $t = 1$, but in an ordered fashion (first e_a, then e_b, and finally e_c). This ensures that the simulation execution and results are repeatable.

3.2.3 VisualSense

VisualSense extends Ptolemy II by introducing the Wireless Director (an extension of the Discrete Event Director), that allows for both wired and wireless connections between the nodes. Typically, wireless connections are used among nodes, and wired connections are used inside the nodes, although other configurations are possible.

In the Wireless Domain, rather than directly linked to relations, ports are bounded to Wireless Channels that can simulate radio communication. Wireless Channels are parameterised in terms of radio signal properties, namely, range, power, propagation speed, loss probability, and propagation factor, that may be overridden by individual ports. When a bounded output port sends a message, the Wireless Channel decides which ports are to receive it, and when, based on the nodes positions, the signal properties, the terrain, and the individual port specifications.

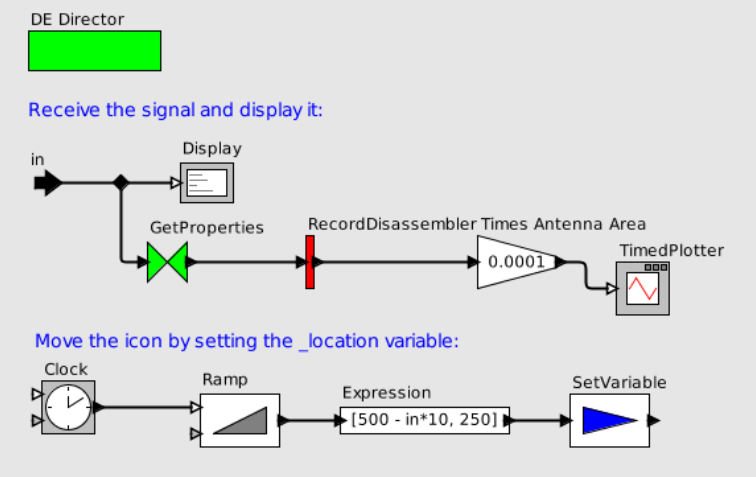

Figure 3.5: A VisualSense sensor node model

VisualSense comes with a library of actors that ease the task of simulation, such as CollisionDetector, LinkVisualizer, GraphicalLocator, and NodeRandomizer. Figure 3.4 shows a network analysis model. On the upper left corner of the model, there is a WirelessDirector, and a NodeRandomizer actor that distributes, at the start of the simulation, the nodes randomly over a given area. At the centre, there is a user defined node called Initiator , that broadcasts the first message. The other nodes around it forward the message, becoming red if they received it directly from the initiator, green if they received it via another node, and staying white if they never receive a message. The model uses two wireless channels: Channel, for the inter-node communication, and ReportChannel, to feed the data analysis actors. Multiple channels can be used to simulate different frequencies, or technologies.

Since VisualSense is a generic WSN simulator, there are no pre-defined sensor node actors. Instead, sensors are modelled from a very high-level, abstract view, using actors. Figure 3.5 shows a model of a node that receives a message on its **in** port, prints it on a GUI window (Display), then retrieves the signal properties as a record (GetProperties), extracts only the signal power (RecordDisassembler), scales it according to the antenna area (with a Scale actor, renamed to Times Antenna Area), and, finally, generates a timed plot of the signal power (TimedPlotter).

Regardless of the operation triggered by a received message, the actor performs another operation. A Clock actor generates periodic events that are used to generate the new position (Ramp and Expression), of the node in the model (actor SetVariable is configured to set the _location attribute of the node).

It is out of the scope of this work to further explore VisualSense's capabilities in terms of terrain modelling, continuous time modelling, or radio communication. VisualSense is a very powerful simulator. Being generic, it enables to model virtually any sensor node, or WSN, at the cost of *having to* define the sensor node, the WSN, and the application it runs.

Chapter 4

The Callas Programming Language

Callas [54] is a programming language for WSN, based on the formalism of a process calculus [50, 54], with the goal of establishing a foundation for developing programming languages and run-time systems for sensor networks. Callas may be used as an intermediate language upon which high-level programming abstractions may be encoded as semantics preserving, derived constructs. The language offers constructs to describe sensor computations, code mobility, and code update.

Callas is *type-safe*, which means that well-typed (Callas) programs do not produce protocol run-time errors. Language type-safety is of utmost importance in WSN, since it allows premature (static) detection of potential errors, thus minimising the amount of debugging required for an application once it is deployed.

The Callas Virtual Machine (CVM) is a stack-based machine that serves as the run-time system for Callas. The CVM guarantees run-time soundness, *i.e.*, that the low-level language (byte-code) it executes preserves the semantics of Callas programs [15].

In Section 4.1, we explain the Callas syntax and semantics along with examples. Section 4.2 presents an overview of the CVM.

4.1 Programming Language

Section 2.4 classifies programming languages for WSN according to three criteria that can now be applied to Callas. In terms of *hardware interaction*, Callas falls on the Virtual Machine category, in terms of *network perception*, it falls on the sensor-based category, and, regarding *data acquisition*, it is communication-centric.

A Callas program is a sequence of terms whose components are type and module declarations, assignments, expressions, and conditionals. Its syntax is line-oriented, with syntactic terms demarcated by the number of spaces in the beginning of a line (indentation), much like in Python's programming language.

The Callas syntax and semantics are introduced here by the example of a temperature monitoring application. Figure 4.1 declares three module types that are used in the

```
# file: types.caltype

defmodule Nil:
  pass

defmodule SenseTemp:
  Nil sample()
  Nil gather(string mac, long time, double temp)

defmodule Deploy:
  Nil deploy(SenseTemp senseTemp)
```

Figure 4.1: Callas type declarations

application. Nil is an empty module type, denoted with keyword **pass** in its declaration; SenseTemp is a module type with two functions, sample and gather. The former has no parameters and returns a Nil module; the latter receives a mac address (as **string**), a time (as **long**), and a temp(erature) (as **double**), returning a Nil module. Finally, the Deploy module type contains the function deploy that receives a SenseTemp module, and returns a Nil module.

```
# file: iface.caltype

from types import *

defmodule Sensor(Deploy, SenseTemp):
  Nil listen()
```

Figure 4.2: A Callas type declaration that serves as a network interface

Type declarations are used to define interfaces. In a Callas network, all nodes share the same (public) interface. This restriction seems plausible because, since nodes collaborate in retransmitting messages from distant nodes to the sink, it is necessary to guarantee, *at compile time*, that messages sent amongst nodes are always understood. In Figure 4.2, the Sensor type declaration extends the Deploy and SenseTemp types, and declares listen, a function with no parameters and Nil return.

Although all nodes share the same interface, they may have different behaviors, according to their role in the WSN. For instance, the sink node in this example application must have a different behaviour from a sensing node because, instead of sensing temperatures, it must log the readings from rest of the network. The two following sub-sections present a Callas program for the sensing device, and another program for the sink device. The details on the abstract syntax, operational semantics, and type system can be found in [54].

```
# file: node.callas

from iface import *

module m of Sensor:
  def listen(self):
    receive

  def deploy(self, code):
    extern logString("Received deployed code.")
    mem = load
    mem = mem || code
    store mem

  def gather(self, mac, time, temp):
    pass

  def sample(self):
    pass

store m

listen() every 500 expire 600000
extern logString("node is listening.")
```

Figure 4.3: A Callas program for a sensing node

4.1.1 A Callas Program for a Sensor Node

The Callas code depicted in Figure 4.3 is meant to be deployed in all network nodes, except for the sink. The program starts by importing the (interface) types. Then, it implements a module of type Sensor, binding it to variable m. Thereafter, it stores module m in memory (**store** m). The language can define timed functions; the expression listen () **every** 500 **expire** 600000 sets up a timer that triggers a call to function listen every 500 time units for a period of 600000 time units. The last expression of the sensor node program calls an external function that logs a message in the sensor's log file. The external expression allows for the interaction between Callas programs and sensor internals: the program passes control to the underlying virtual machine and expects a result (in this case the program disregards the result from the call). The type system checks that the external functions are called correctly, *i.e.*, that the node's native function signatures and return types are observed.

All functions receive, as first parameter, a reference to the module where the call is being made: **self**. This argument is passed automatically by the run-time system and allows functions to call other functions in its own module, or themselves recursively.

The functions defined in module m (of type Sensor) are now detailed. Each node is equipped with an incoming and an outgoing queue for interaction with the network. Function listen executes a **receive** expression that explicitly takes a message from the node's

incoming message queue and places it in the CVM's run queue; the program continues executing even if there are no incoming messages. Function deploy receives a module and installs it in the node's memory. For that, it loads the node's memory to a variable mem, using expression **load**, updates the module by joining it with the code received as an argument, and stores the updated module back into the node's memory using expression **store**. The module joining operation (mem || code) creates a new module that contains all functions from module mem replaced by the correspondent function from module code. For the functions in code that are not a member of mem, the joining operator simply adds them to the resulting module. Functions gather and sample are left empty.

4.1.2 A Callas Program for a Sink Node

Similarly to the sensor program, the sink program (Figure 4.4) starts by importing the interface and implementing the Sensor module. However, it provides a different implementation. The deploy function is empty, since the sink does not install code remotely from the network, the sample function broadcasts (using the **send** expression) a sample remote call, and gather logs the received messages.

The program then defines module toDeploy, of type SenseTemp, intended to be deployed in the network. Functions gather and sample, once module toDeploy is installed on the sensing nodes, replace the corresponding functions in the sensor program. The implementation of gather and sample could be provided in the sensor program up front, it is provided here to demonstrate code deployment in Callas.

In module toDeploy, the gather function broadcasts a call to gather with the same arguments it receives, thus eventually routing values from distant nodes to the sink; function sample uses external functions to get the node MAC address, the current time, and the current temperature. It then calls gather in order to broadcast the node values.

In the last part of the program, a call to deploy with module toDeploy as argument is broadcasted, redefining the network application. Finally, timers to listen and sample are set up. The former periodically places incoming messages on the CVM's run queue, while the latter "queries" the network, possibly resulting in gather messages coming to the sink.

4.1.3 Callas Network Application

A Callas network application may be composed of several files. In this Chapter's example application there are five: *i)* types.caltype, that defines the Callas types, *ii)* iface.callas, that defines the (public) functions available for each sensor, *iii)* node.callas, a file with the Callas program for the sensor nodes, *iv)* sink.callas, a file with the Callas program for the sink node, and *v)* network.calnet (in Figure 4.5), that describes the network, by specifying its interface and programs. The compilation of this calnet file results in files node.calbc and sink.calbc, respectively, the byte-code versions of node.callas and sink.callas.

```
# file: sink.callas

from iface import *

module m of Sensor: # make sure we have an initial memory
  def listen(self):
    receive

  def deploy(self, tempSense):
    pass

  def sample(self):
    send sample()

  def gather(self, mac, time, temp):
    extern logString(mac)
    extern logLong(time)
    extern logDouble(temp)
    extern logString(" ")

store m

module toDeploy of SenseTemp:
  def gather(self, mac, time, temp):
    send gather(mac, time, temp)

  def sample(self):
    mac = extern macAddr()
    time = extern getTime()
    temp = extern getTemperature()
    send gather(mac, time, temp)

extern logString("deploying module.")
send deploy(toDeploy)
listen() every 500 expire 600000
extern logString("trigger sample.")
sample() every 30000 expire 300000
extern logString("waiting for gather.")
```

Figure 4.4: A Callas program for a sink node

```
# file: network.calnet

interface = iface.caltype # network's interface

sensor:
  code = node.callas      # program for the node

sensor:
  code = sink.callas      # program for the sink
```

Figure 4.5: A Callas network program

4.2 Virtual Machine

The Callas Virtual Machine is a stack-based machine that serves as the run-time system for Callas. The machine state is divided into:

- an internal clock $\mathcal{Q}$ that uses arbitrary time units t;
- a module $\mathcal{M}$ that represents the shared memory of the device. The code of its functions can be altered during the execution of the device, but, since all nodes must always adhere to the network interface, functions cannot be added or removed;
- a set of programmed timed function calls $\mathcal{T}$;
- a call-stack $\mathcal{C}$ whose components, called call-frames, are divided into a program counter, an environment frame $\mathcal{E}$, an operands stack $\mathcal{S}$, a byte-code $\mathcal{B}$, and a constants array $\mathcal{U}$;
- a (run-)queue of pending calls $\mathcal{R}$.
- an incoming and an outgoing queues $\mathcal{I}/\mathcal{O}$ of serialised calls that enable interaction with the lower layers of the network protocol.

CVM executes a low-level language (byte-code) that defines an instruction set. There are instructions for manipulating modules, making calls, moving data, network input/output, control-flow, and basic arithmetic and logic operations. The instruction set, byte-code, and CVM's internals are not in the scope of this work and can be found in [15].

Chapter 5

Simulation of Callas Applications

This chapter presents an approach towards the generation of simulation models. It consists of creating VisualSense simulation models from Callas applications. The first step in this approach is to embed the Callas Virtual Machine (CVM) in VisualSense, configuring it, and adapt its interface in a VisualSense actor. Besides input/output message handling, this step comprises devising means of parameterising the CVM clock with the simulation clock, and of dealing with native operations. The second step is to define generic network and sensor model templates that serve as configurable, extensible building blocks for Callas simulation models. The third (and last) step is to create a model generator. We create an extensible characterisation of WSN that conveys information needed for the automatic simulation models generation.

Section 5.1 addresses the subject of embedding the CVM in a VisualSense actor. Section 5.2 presents generic network and node simulation models, and, Section 5.3, presents the simulation model generator. Finally, Section 5.4 addresses the performance and scalability of the generated simulation models.

5.1 The Callas Virtual Machine as a VisualSense Actor

The Callas programming language follows the event-oriented programming paradigm, therefore, it is natural that, in VisualSense, an event-oriented computation domain is a more suitable execution environment for an actor that adapts the CVM. VisualSense's Discrete Event domain makes use of a simulation clock and of a (time-ordered) queue with (time-stamped) events to simulate event-oriented systems.

In order to adapt the CVM in a discrete event actor, it is necessary to assess how the CVM state is affected during the computation. The CVM state (introduced in subsection 4.2) can be divided into two parts: *i) clock and network layer*, comprising precisely a clock $\mathcal{Q}$ and a network interface $\mathcal{I}/\mathcal{O}$, is the part of the state that can be directly altered by the simulator, and *ii) machine state*, encompassing a call-stack $\mathcal{C}$, a run-queue $\mathcal{R}$, a set of timers $\mathcal{T}$, and a collection of modules $\mathcal{M}$, is the part of the state that is isolated

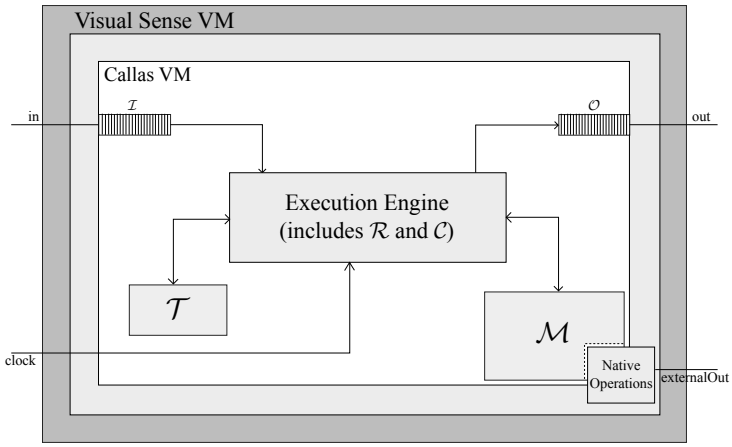

Figure 5.1: Configuration of VisualSenseVM and of the adapted CVM

from the simulator and that interacts with the clock and network layer.

The CVM is parameterised by a native operations object that wraps the physical device operations. Some native operations have side-effects (*e.g.*, turning on a LED), while others do not (*e.g.*, getting the node MAC address). To simulate a given device, it is necessary to create an object that is able to mimic the device native operations in VisualSense (including side-effects), and parameterise the adapted CVM with it.

Given a clock and a network layer, the machine state is agnostic of the execution environment and of the native operations. In fact, the native operations module is not a special module among $\mathcal{M}$, since, like other modules, it abides to an interface, and that the side-effects of its operations are not known to the internal state.

The CVM is adapted in an actor called VisualSenseVM, parameterised with a CVM, a native operations object, and the path to a Callas byte-code file. Figure 5.1 represents the VisualSenseVM actor internal configuration and the adapted CVM.

The VisualSenseVM actor has two input ports and two output ports. Ports in (input) and out represent the network interface of the CVM. Port clock (input) is used to parameterise the execution engine with the simulation time. Finally, port externalOut is an output port that is used by the native operations object to externalise the native operations side-effects.

VisualSenseVM is an adapter that converts the messages received in its in port to the CVM message format and places them in the CVM input queue, $\mathcal{I}$. Conversely, the messages on the CVM output queue $\mathcal{O}$ are converted to Ptolemy II messages and sent to the actor out port. The CVM execution engine is then responsible for retrieving messages from $\mathcal{I}$ and placing messages on $\mathcal{O}$. The strategy for externalising the native operations consists of encoding side effects as messages sent to the externalOut port, thus delegating the simulation of side-effects to specialised actors that handle such messages.

The CVM must keep pace with the simulation clock. If VisualSense takes one minute

(of simulation time) to simulate one second (of simulated time), then the CVM must be synchronised with the simulator clock, and not with the hardware platform clock, otherwise, the CVM would be, as far as VisualSense is concerned, running in the future, because its internal clock would run 60 times faster than the simulation clock. One way to synchronise the CVM with the simulator clock is to parameterise time. Instead of executing instructions based on its own clock ticks, the CVM execution engine receives tokens (simulating ticks) from an external clock. The external clock is a VisualSense actor, synchronised with the simulator clock by design, and is able to send a parameterisable number of tokens per second, which can be used to simulate nodes running at different clock rates. Also, clocks can be configured to start ticking at distinct times, yielding nodes that start executing at distinct times.

The VisualSenseVM actor controls the CVM execution. When a token is received in the clock port, simulating a hardware clock tick, VisualSenseVM performs the following actions (in this order):

- Tells the Callas VM to run a computation step (a single operation) based on the current simulation time. Timed calls ($\mathcal{T}$) are managed by the execution engine and are, therefore, also triggered according to the simulation clock;
- If there is a message for an external operation, converts it to the Ptolemy II format and sends it to the externalOut port;
- If there is a message on the Callas VM output queue, converts it to the Ptolemy II format and sends it to the VisualSenseVM out port.

The computation step is performed first because it may result in a new message on the CVM output queue that must be sent to the out port, or in a new message from the native operations object that must be sent to the externalOut port, in both cases, at the current simulation time. Since the CVM runs a single step per clock tick, the VisualSenseVM gets at most one message from the CVM, therefore, the order of message handling (the last two actions) is irrelevant.

5.2 Network and Node Simulation Models

The models presented in this section are building blocks for simulation model generation. The simulation as a hole is parametric in a model that defines the network, a model for each type of sensor, and specific model parameters. The network model defines a container for node actors and communication channels. Each node is, in itself, a simulation model of a physical device running Callas. Both the network and sensor models can be configured or extended. The network model, for instance, may also comprise environment models, or auxiliary actors that can perform network analysis.

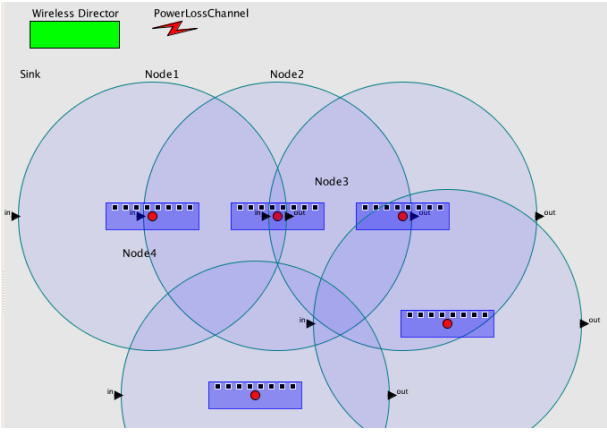

Figure 5.2: A VisualSense model of a network containing five nodes

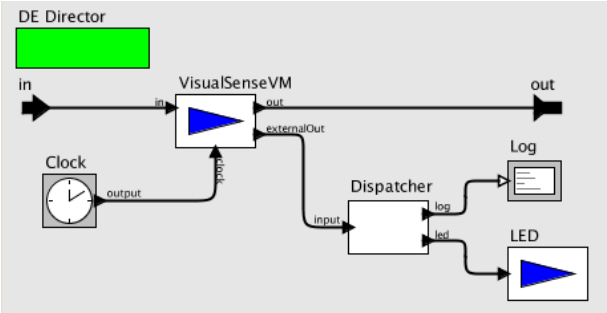

Figure 5.3: A generic VisualSense model of a node

5.2.1 Network Model

The (basic) network model consists of a Wireless Director and a Wireless Channel. In Figure 5.2 depicts a network model consisting of a WirelessDirector, a PowerLossChannel, and five nodes. Each node is represented in the GUI by a (blue) box with a (red) spot on the centre marking its exact position, and a row of (black) squares, modelling a LED array. The wide (light-blue) circle around the node represents its transmission range. In this representation, it is possible to perceive that the lowermost node is isolated from the other nodes: its transmission range is insufficient to reach any of them, and the others cannot reach it as well.

5.2.2 Node Model

A node has an input port in, and an output port out, bound to the wireless channel, simulating signal transmission. The node model (depicted in Figure 5.3) is composite; it contains a set of (atomic) actors that define its behaviour. Next, we detail each actor.

Clock

Is a Ptolemy II actor that simulates a hardware clock. It is parameterised by the tick period value. Setting the tick period to 0.01 seconds, for instance, yields a 100 Hz hardware clock. Hence, this actor sends one hundred tokens trough its output port, per simulation second. Each sensor has its own clock, allowing to simulate a network where nodes run at different clock rates.

VisualSenseVM

The VisualSenseVM actor, detailed in Section 5.1, adapts the CVM by managing its network interface, by providing an implementation of the native operations for the node, and by controlling the execution engine via the Clock actor.

Dispatcher

Dispatches messages coming from the VisualSenseVM externalOut port to the actors that simulate native operations with side-effects. The input port accepts record tokens in the form "operation": opToken, "arguments": argsToken that map an operation to its arguments. For each token received, the Dispatcher breaks the record and sends the argsToken to an output port that has the same name as the value of the opToken.

This actor can be extended in order to support any native operation. It suffices to create a new output port, with the same name as the operation, and set its type accordingly to the argsToken. This can be accomplished with little effort using the VisualSense GUI.

LED

Responsible for performing the operations needed to update the sensor graphical representation in a manner that simulates turning LEDs on and off.

Log

Prints log entries to a GUI window. The Log actor can be replaced by another actor that can perform something more elaborate, such as writing to a file, or feeding a database.

Summarising the node actor, VisualSenseVM adapts the CVM, thus holding the program memory and acting as a processing unit, running at a clock rate given by the Clock actor. VisualsenseVM parameterises the CVM with a native operations object that mimics a given device native operations and routes the messages representing side-effects to the externalOut port. These messages are then sent by the Dispatcher to specialised actors that simulate side-effects (*e.g.*, LED). This model can be extended in order to support virtually any sensor device by simply providing a native operations implementation, the corresponding set of native operation actors, and configuring the Dispatcher.

5.3 Automatic Generation of Simulation Models

This section addresses the simulation model generator tool and an extensible characterisation of WSN simulation, which is used to automatise the model generation.

Callas applications are defined in calnet files. A calnet file is a mapping of parameters to arguments, where the argument may either be a single value, or a mapping of keys to values. Figure 4.5 presented a calnet file for the temperature monitoring application example of Section 4. The information in that file is used to compile and type check the network. Given that the Callas compiler ignores the parameters it does not need, the same file can be used to convey a WSN characterisation that enables automatic model generation. Such characterisation, piggybacked on the calnet file, includes indicating how many nodes (and of which types) compose the network, their positionings, communication range, etc.

The characterisation consists of a set of *generator parameters* that may be of two types: *i)* simulation parameters, mandatory, specified in the form parameter = value, and *ii)* actor parameters, optional, specified in the form ActorName.parameter = value. Figure 5.4 depicts a calnet file, containing generator parameters. This file is an extended version of the one presented in Figure 4.5, from which only three lines (identified in the Figure) are taken. From such file, the simulation model generator tool creates a MoML (persistent format of Ptolemy II models, introduced in Chapter 3) file. The generator parameters may also be global, when declared for the network as a whole (outside sensor definitions, like template = network.moml), or local, when declared for specific groups of nodes (inside sensor definitions, like Clock.period = 0.001).

The second parameter template, configures the generator tool to use the network.moml as the model for the network. The third, CallasPowerLossChannel.lossProbability (optional), redefines the lossProbability parameter of the CallasPowerLossChannel, an extension of the PowerLossChannel that eliminates repeated messages on the network.

Each sensor definition encloses a set of parameters that determines the common properties of a group of nodes. The size parameter tells the generator tool how many nodes run the specified code, and the template is used to get the node MoML model. The range parameter determines the node's transmission range and the the position specifies how they are distributed in the field. It can have three forms:

- explicit x_1, y_1 ... [x_n, y_n], explicitly defines the positions for all the nodes;
- uniform x_1, y_1 to x_2, y_2, uniformly distributes nodes within a bounding box defined by the arguments;
- random x_1, y_1 to x_2, y_2, like the former, but with a random distribution.

The simulation model generator tool does not have any hard-coded models. It creates simulation models from calnet files decorated with (mandatory) simulation parameters.

```
# file: networkReadyForSimulation.calnet

interface = iface.caltype  # from network.calnet

template = network.moml
CallasPowerLossChannel.lossProbability = 0.0

sensor:
  code = node.callas  # from network.calnet
  size = 100
  range = 250
  position = random 0,0 to 1000,1000
  template = genericNode.moml
  Clock.period = 0.001

sensor:
  code = sink.callas  # from network.calnet
  size = 1
  range = 5000
  position = explicit 0,0
  template = genericNode.moml
  Clock.period = 0.001
```

Figure 5.4: A Callas network file extended with generator parameters

The (optional) actor parameters are not predefined. The generator tool will search for actors whose names match the actor name in actorName.parameter = value and set the parameter's value accordingly, therefore, actor parameters provide a means of configuring any actor defined (at any hierarchical level) in the models. These are design decision that we believe make the generator tool very flexible.

5.4 Performance and Scalability

For a fixed number of nodes, the performance and scalability of a simulation may vary greatly accordingly to the number of messages exchanged and to the complexity of the signal, terrain, and physical phenomena being modelled. It is out of the scope of this work to evaluate simulation models for a set of WSN use cases.

We choose the model generated from networkReadyForSimulation.calnet to evaluate performance and scalability. This model does not account for physical phenomena or terrain modelling, and uses a simple signal without loss or collision. Furthermore, it uses an unrealistic temperature model. However, our goal is to evaluate the aforementioned requirements regarding the number of nodes and number of messages exchanged. Naturally, more complex and "real" models will yield simulations with lower performance and scalability.

Other MoML models were created for the same Callas application, but with larger network sizes, sink ranges, and wider node distributions. Table 5.1 lists the number of

Table 5.1: Measurings of simulation duration and memory footprint

Nodes	`moml` (MB)	Duration (m)	Memory (MB)
100	1.0	3.5	250
200	2.2	12.48	228
300	3.4	17.88	220
400	4.7	28.90	259
500	5.9	43.47	307
600	6.9	52.02	314
700	7.9	60.18	371
800	8.9	68.95	418
900	9.9	77.63	441
1000	10.9	85.94	464
2000	21.8	3.80	870
3000	32.7	6.65	1440
4000	43.6	9.97	2140
5000	54.5	15.9	2550

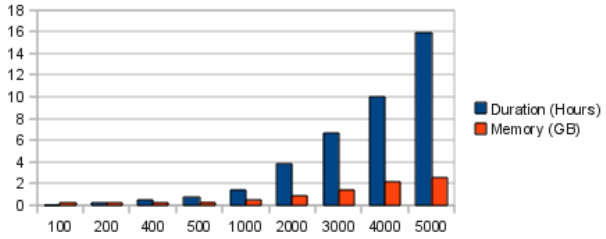

Figure 5.5: Simulation duration and memory footprint

nodes, MoML file size (in MB), simulation duration (in minutes), and memory footprint (also in MB) for the different networks. Figure 5.5 plots the same values in a graph. The results were obtained with VisualSense 7.01 on a Linux based PC with an Intel QuadCore 2.66GHz CPU and 3.4GB of RAM. We did not take into account the amount of time it takes to generate such models, since it is negligible.

In terms of scalability, our experiments show that simulation duration has a quadratic growth, while memory footprint grows linearly. We believe that simulation duration is not a critical factor, as one would expect to wait for a few hours before having results for a 5000 node network.

Memory footprint is very important since it limits the network size. At this point no optimisations were made, so we expect to reduce the network size to memory footprint ratio, enabling to simulate even larger sensor networks in an out-of-the-shelf computer.

Chapter 6

Integrating WSN Simulation into Workflow Testing and Execution

One of the latest application areas of sensor networks is the *Internet of Things and Services* (ITS), which aims at integrating the *state of the world* seen from the *eyes* of sensors in high-level applications available on the *Internet*. An ITS application may benefit from environment observations and tailor its behaviour accordingly. For example, a home automation system may extend its features and, in addition to the traditional task scheduling (*e.g.*, turning on and off a given device according to a predetermined plan), react to environmental conditions or to behavioural rules of the inhabitants. Another application area is logistics, where, for example, we may want to alter a delivery process according both to the actual conditions of the goods being transported and to the traffic information in the route to the destination. In this case, the information obtained from the sensors can lead to modifications in the delivery process, such as a change in the order by which the goods are delivered.

Applications that encode workflows are difficult to test because their behaviour depends on external events (the world) that are, in the general case, nondeterministic. For workflows that involve WSN, simulation may ease testing and execution by allowing to separate the WSN concern from the workflow, and to provide fresh data for the workflow.

This Chapter presents a scenario in logistics that illustrates both an application for WSN simulation, and the use of workflow management systems. The VisualSense simulation of the scenario is integrated with a workflow management system, providing a means of integrating WSN data into the workflow execution and testing.

6.1 A Logistics Scenario

The scenario used in this chapter describes a workflow process integrated with environmental readings obtained from a WSN. A truck departs from Lisbon with two vaccine containers, each equipped with a temperature sensor. In both containers the temperature should not exceed $10^{\circ}\mathrm{C}$, otherwise the quality of the vaccines may be compromised.

```
# file: types.caltype

defmodule Nil: pass

defmodule AlertTemp:
  Nil sample()
  Nil alert(string mac, long time, double temp)
```

Figure 6.1: Callas types for a logistics application

```
# file: iface.caltype
from types import *

defmodule Sensor(AlertTemp):
  Nil listen()
```

Figure 6.2: Interface type for a logistics application

The first container is to be delivered in Coimbra and the second in Porto. The truck is equipped with a base station that periodically polls the sensors for the temperature in the containers and informs the control centre, via GSM, if any reading exceeds $10^{\circ}\mathrm{C}$. In the control centre a new workflow is initiated when a communication from the truck is received, possibly determining changes to the delivery order. Testing this workflow using the traditional approach requires a method to replay the communications with the truck. However, this is not required if the simulation of sensor networks is integrated with the workflow execution. Furthermore, the integration may allow the use of simulation data from sensor networks in richer scenarios than the one just described.

An implementation of the WSN application for this scenario is presented to further introduce Callas and demonstrate simulation model generation. Figures 6.1 and 6.2 depict the Callas types and network interface for the application. All nodes in the network share the same type, Sensor, that extends type AlertTemp by adding function listen.

The file node.callas (Figure 6.3) defines the program for the nodes, that is, for the sensors in the containers. The program imports the interface iface.caltype, defines module M that implements Sensor, installs it in the sensor memory (**store** m), and schedules the periodic execution of function listen. Module m implements *i)* function listen for reading messages from the input queue (**receive**), *ii)* function sample, that emits a network call to function alert (mac, time, temp), with the sensed values, and *iii)* function alert, empty.

Program sink.callas (Figure 6.4) implements the same interface, but with a different behaviour from the program running on the sensors. In the sink, function Sample only sends to the network a call to sample(). Function alert records the values received and, if the temperature raises above $10^{\circ}\mathrm{C}$, sends a message by GSM to the control centre. In addition to the periodic scheduling to function listen, it also schedules function sample, which polls the temperature values from the network.

```
# file: node.callas
from iface import *

module m of Sensor:
  def listen(self):
    receive

  def sample(self):
    mac = extern macAddr()
    time = extern getTime()
    temp = extern getTemperature()
    send alert(mac, time, temp)

  def alert(self, mac, time, temp):
    pass

store m
listen() every 30000 expire 36000000
```

Figure 6.3: Sensing node program for a logistics application

```
# file: sink.callas
from iface import *

module m of Sensor:
  def listen(self):
    receive

  def sample(self):
    send sample()

  def alert(self, mac, time, temp):
    extern logString(mac)
    extern logLong(time)
    extern logDouble(temp)
    extern logString(" ")
    if temp > 10.0:
      extern sendGSM(mac, time, temp)

store m
listen() every 30000 expire 36000000
sample() every 30000 expire 36000000
```

Figure 6.4: Sink node program for a logistics application

```
# file: network.calnet

interface = iface.caltype

template = containerNetwork.model

sensor:
  code = node.callas
  size = 2
  range = 50
  position = random 0, 0 to 10, 10
  template = containerSensor.moml

sensor:
  code = sink.callas
  size = 1
  range = 50
  position = explicit 0, 0
  template = gsmSink.moml
```

Figure 6.5: Callas application for a logistics scenario

The simulation model for the logistics WSN application is generated from the file network.calnet (Figure 6.5).

6.2 Workflow Execution Integrating WSN Simulation

Workflow testing and execution is aided with specialised computer systems called workflow management systems. These systems, such as YAWL System [70], are mainly used to analyse business processes, but can also be used for design-time business process validation [63].

Scientific workflow management systems are a specialisation of the more general type that allow to execute scientific workflows (for instance, a structured set of operations over environment measurement data). Among such systems are Taverna [34], Triana [67], and Kepler [5]. Kepler supports GUI modelling and execution, workflow composition, distributed computation, and access to data repositories and web services. Like VisualSense, Kepler is a Ptolemy II specialisation.

In Kepler, the workflow execution is determined by a computation domain. For example, in the Synchronous Dataflow domain, the execution is synchronous and occurs in a pre-calculated sequence; in the Process Networks domain it is parallel, meaning that one or more components may run simultaneously; and, in the Discrete Event domain, workflow execution is triggered by events and takes time into account.

Kepler and VisualSense may use the same components interchangeably. Therefore, it should be possible to include simulation models from VisualSense in Kepler workflows, obtaining a means of integrating information from the sensor network simulation into the

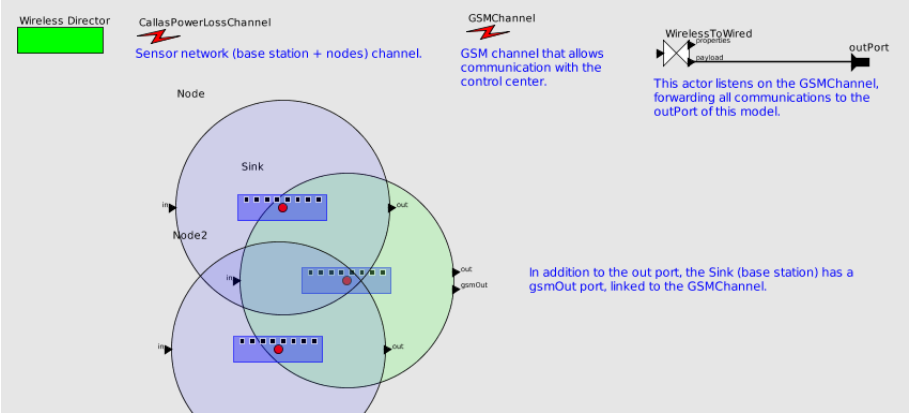

Figure 6.6: Simulation model to be integrated in Kepler

workflow execution. Furthermore, we are able to generate VisualSense simulation models from Callas applications, which eases all the simulation-related work from the workflow tester/executor perspective.

6.2.1 Integrating WSN Simulation in Kepler

Section 6.1 presented an environment monitoring application for a WSN. From that application (defined in file network.calnet), a simulation model is automatically generated. The network template used allows the integration of the model as an actor in another model defining an output port for the model.

Figure 6.6 depicts the generated model. The two temperature sensors, Node1 and Node2, in blue, execute the code in node.callas (Figure 6.3). Periodically, they send the temperature in each container to the base station, represented in green. The base station (sink) executes the code in sink.callas. It is possible to identify the nodes communication ranges, as well as their relative positions.

The communication between the sensor nodes and the base station is made through channel CalasPowerLossChannel. Ports in/out of all the represented nodes receive/send messages through CalasPowerLossChannel. The communication of the base station with the control centre is simulated by the GSMChannel. The base station is the only node that can send messages on GSMChannel, namely output messages, making use of its gsmOut port.

The workflow described in Figure 6.7 calculates the best path for the deliveries, taking into account the container temperatures, and the truck location. The sensor network integration is made by encapsulating the network model in a Kepler actor (TruckNetwork), that acts as a data source. The workflow execution is, in this case, triggered by a message received from TruckNetwork. In a different workflow, the execution could be started by another event, possibly being altered by an incoming message from the sensor network. In the top right corner of Figure 6.6, there is the WirelessToWired actor, that re-

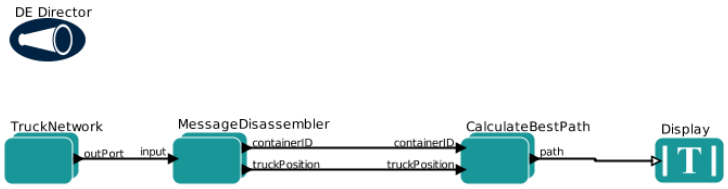

Figure 6.7: Kepler workflow where TruckNetwork encapsulates the WSN model

ceives the messages sent through GSMChannel and dispatches them through the outPort of TruckNetwork, depicted in Figure 6.7. In the workflow integration, this message is routed to a MessageDisassembler actor that extracts the containerID and truckPosition values, needed for the CalculateBestPath workflow. It should be mentioned that the WirelessToWired actor (that connects the sensor network to the workflow) is independent of the network and of the workflow; it is only a message broker.

The difficulties of integrating VisualSense's sensor network models in Kepler lie in the (possible) computation domain heterogeneity. WSN simulation is done in the Wireless domain, an extension of the Discrete Event domain that is not suited for all types of workflows. However, there should be no difficulties in the type of business processes that we are interested in simulating, because they are usually event oriented.

6.2.2 Workflow Interoperability

A way to mitigate integration difficulties, and to use WSN simulation models in more workflow management systems is to create robust mechanisms of workflow interoperability, *i.e.*, the ability to execute workflows in a distributed way, using two or more distinct workflow management systems. The available variety of workflow management system engines and description languages hinders workflow interoperability. For example, Taverna [34] interprets the Simple Conceptual Unified Flow Language [34] (SCUFL), Kepler uses the Modelling Markup Language [12] (MoML), Yawl system [70] uses Yet Another Workflow Language [69] (YAWL), and Triana [67] interprets, in addition to its proprietary format, the Business Process Execution Language [40] (BPEL). Summing to the aforementioned difficulties (the variety of description languages and of execution platforms), workflow specification languages usually have different degrees of expressiveness, making the translation among them not always possible, which compromises interoperability by language translation.

Workflow interoperability could be achieved by standardising the specification/execution language. There has been such attempts, for example, the Workflow Management Coalition created XPDL [40], and Microsoft and IBM have created BPEL, both aiming to become *the standard*. Another way to achieve interoperability of workflows is by inte-

grating workflow engines so that it is possible to run each workflow in its execution environment, but being able to interact with other workflows, running in other environments. Such an approach is proposed by Kukla et al. [41]. The authors see workflow management systems as (legacy) applications embedded in a Grid Computing environment, in the case, GEMLCA [16] (Grid Execution Management for Legacy Code Applications).

Chapter 7

Conclusion

Wireless Sensor Networks are becoming an increasingly important topic as the number (and complexity) of their applications increase. The advances in hardware and software for WSN allow for cheaper devices, improved energetic autonomy, greater computational capabilities, efficient communication protocols, and higher-level programming models. It is often stated that WSN will become ubiquitous in a very near future, since they can be an enabling factor for new services, such as for the *Internet of Things and Services*. In spite of the advances in this field, deployment is still one of the main difficulties regarding WSN, notably because testing and debugging WSN applications is hard.

Simulation allows us to define virtual environments where experimental deployments can be done, tested, and analysed. Therefore, it can play an important role in testing and debugging. However, defining simulation models is a laborious task. This thesis addresses that problem by proposing automatic generation of simulation models for WSN directly from the source code of WSN applications. In particular, we generate simulation models for the VisualSense simulator, from Callas applications. For that purpose, we *i)* adapt the Callas Virtual Machine in a VisualSense component (actor), *ii)* define simulation models for generic WSN and devices, and *iii)* create a model generator tool that can be parameterised in a very complete and flexible way. In particular, we can parameterise not only the models to be used for the network and each type of node, but also each individual actor therein. The generated models may, for instance, have distinct nodes, possibly running distinct applications (although Callas ensures that they abide to the network interface), with different clock ratios, be turned on and off anytime during simulation. Our approach for simulating Callas WSN has good performance and scalability, as depicted in Figure 5.5. We are able to simulate networks with several hundreds of nodes using out-of-the-shelf hardware.

This thesis also presents an approach to integrate sensor network simulations into the execution of workflows (in workflow management systems), thus providing a way of testing higher level applications based on information from *things* in the world. Our proposal integrates the simulation of sensor networks of VisualSense [7] in the Kepler

workflow management system [5], exploiting the interoperability of components (actors) between the two systems. This approach is feasible because both systems are extensions of the Ptolemy II modelling and simulation platform [23]. We believe that our approach is a valuable contribution for testing applications (not only those based on workflows) that depend on environmental values.

We foresee that this work can be extended in several different ways. A possible extension regards the simulation model generation, it is our goal to extend the network and sensor models (with the ability to account for battery models, for example) in order to further ease network and sensor template model definition; this will also allow us to measure network lifetime and to experiment with energy conservation. Another extension impacts the performance of the model. We anticipate that two facts contribute definitely for such optimisation: the reduction of the generated model file size and the decrease of the Callas Virtual Machine actor memory footprint, that would also improve the scalability of the simulation.

Regarding the integration of sensor network simulation in workflow testing and execution, our initial focus will be on the validation of the proposed integration model, as well as on obtaining results that allow us to evaluate the proposed solution. Another interesting point that deserves further attention is workflow interoperability, *i.e.*, the ability to execute workflows in a distributed way, using two or more distinct workflow management systems. As discussed in Chapter 6, we intend to investigate the possibility of integrating distinct workflow engines in a Grid Computation system that may serve as a meta workflow management system. This would allow us to extend the integration of WSN simulation to other workflow management systems.

Finally, another direction for future work concerns with the availability of sensor information via web, this does not seem complicated and we envision what it could be easily achieved using web services: we would need to create an interface that exposes the VisualSense simulator as a web service so that it can be accessed remotely.

Bibliography

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 23th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] I. Akyildiz and J. Jornet. Electromagnetic wireless nanosensor networks. *Nano Communication Networks*, 1(1):3–19, 2010.

[4] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[5] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM'04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 423–424. IEEE Computer Society, 2004.

[6] J. Ayres and S. Eisenbach. Stage: Python with Actors. In *International Workshop on Multicore Software Engineering (IWMSE)*, May 2009.

[7] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modelling of sensor nets in Ptolemy II. In *Proceedings of IPSN'04*, pages 359–368. ACM Press, 2004.

[8] J. Barros. *Learning from Data Streams - Processing Techniques in Sensor Networks*, chapter 2. Springer-Verlag, 2007.

[9] J. Beutel, M. Dyer, A. Meier, M. Wöhrle, and M. Yücel. Fast-Prototyping of Wireless Sensor Networks. In *INSS 2007*, pages 1–81. Universität Karlsruhe, Fakultät für Informatik, 2008.

[10] 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. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[11] A. Boulis, C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, pages 187–200. ACM Press, 2003.

[12] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2008-28, Electrical Engineering and Computer Sciences University of California at Berkeley, 2008.

[13] J. Cao, W. Clevel, and D. sun. The s-net system for internet packet streams: Strategies for stream analysis and system architecture. *Journal of Computational and Statistical Graphics: Special Issue on Streaming Data*, 2003.

[14] T. Carley. Sidh: A wireless sensor network simulator. Technical report, Department of Electrical & Computer Engineering University of Maryland, 2004.

[15] T. Cogumbreiro, P. Gomes, F.Martins, and L. Lopes. Safe-by-design programming languages for wireless sensor networks. *ACM Transactions on Programming Languages and Systems (to appear)*, 2010.

[16] T. Delaitre, A. Goyeneche, P. Kacsuk, T. Kiss, G. Terstyanszky, and S. Winter. Gemlca: Grid execution management for legacy code architecture design. In *EUROMICRO'04: Proceedings of the 30th EUROMICRO Conference*, pages 477–483. IEEE Computer Society, 2004.

[17] dimap. dimap web site. `http://www.dimap.es`.

[18] dimap. Sisvia web site. `http://www.dimap.es/enviromental-agriculture-services.html`.

[19] C. Duarte and C. Talcott. Clara: An actor language for high performance distributed computing. In *SBAC-PAD'2000*, 2000.

[20] P. Dubois, C. Botteron, V. Mitev, C. Menon, P.-A. Farine, P. Dainesi, A. Ionescu, and H. Shea. Ad-Hoc Wireless Sensor Networks For Exploration Of Solar-System Bodies. *Acta Astronautica*, Volume 64(Issues 5-6):470 – 478, 2009.

[21] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Workshop on Embedded Networked Sensors*, November 2004.

[22] E. Egea-Lopez, J. Vales-Alonso, A. Martinez-Sala, P. Pavon-Mariño, and J. Garcia-Haro. Simulation Scalability Issues in Wireless Sensor Networks. *IEEE Communications Magazine*, 44(7):64–73, 2006.

[23] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of IEEE*, 91(2):127–144, Jan 2003.

[24] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: An energy-aware resource-centric rtos for sensor networks. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 256–265. IEEE Computer Society, 2005.

[25] P. A. Farrington, H. Nembhard, D. Sturrock, G. Evans, and X. Chang. Network simulations with opnet.

[26] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662. IEEE Press, 2005.

[27] W. F. Fung, D. Sun, and J. Gehrke. COUGAR: The network is the database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'02)*, 2002.

[28] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[29] L. Girod, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *In Proceedings of the 2004 USENIX Technical Conference*, pages 283–296, 2004.

[30] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming Wireless Sensor Networks using Kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, June 2005.

[31] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence . *International Joint Conference on Artificial Intelligence*, 1973.

[32] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM Press, 2000.

[33] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (ENSS'04)*, pages 81–94. ACM Press, 2004.

[34] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server issue):729–732, 2006.

[35] IEEE. Ieee 802.15.1. `http://www.ieee802.org/15/pub/TG1.html`.

[36] IEEE. Ieee 802.15.3. `http://www.ieee802.org/15/pub/TG3.html`.

[37] IEEE. Ieee 802.15.4. `http://www.ieee802.org/15/pub/TG4.html`.

[38] K. Jensen, J. Weldon, H. Garcia, and A. Zettl. Nanotube radio. *Nano Letters*, 7(11):3508–3511, 2007.

[39] Anna N. Kim, Fredrik Hekland, Stig Petersen, and Paula Doyle. When hart goes wireless: Understanding and implementing the wirelesshart standard. In *ETFA*, pages 899–907. IEEE, October 2008.

[40] R. Ko, S. Lee, and E. Lee. Business process management (bpm) standards: A survey. *Business Process Management journal*, 15(5), 2009.

[41] T. Kukla, T. Kiss, G. Terstyanszky, and P. Kacsuk. A general and scalable solution for heterogeneous workflow invocation and nesting. In *WORKS 2008: Proceedings of the Workflows in Support of Large-Scale Science, Third Workshop*, pages 1–8. Springer-Verlag, 2008.

[42] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85–95. ACM Press, 2002.

[43] P. Levis, D. Gay, and D. Culler. Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines. Technical Report UCB//CSD-04-1343, University of California at Berkeley, August 2004.

[44] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of SenSys 2003*. ACM Press, 2003.

[45] M. Li and Y. Liu. Underground structure monitoring with wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 69–78, New York, NY, USA, 2007. ACM.

[46] Libellium. Waspmote web site. `http://www.libelium.com/products/waspmote`.

[47] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services (MobiSys'04)*, June 2004.

[48] L. Lopes, F. Martins, and J. Barros. *Programming Sensor Networks*, chapter 3. Springer-Verlag, 2008.

[49] L. Lopes, F. Martins, and J. Barros. *Middleware for Network Eccentric and Mobile Applications*, chapter 2, pages 25–41. Springer-Verlag, 2009.

[50] L. Lopes, F. Martins, M. Silva, and J. Barros. A calculus for programming wireless sensor networks. In *Proceedings of SENSORCOMM'07, International Conference on Sensor Technologies and Applications*, IEEE Press, pages 451–456. IEEE Press, 2007.

[51] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[52] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[53] M. Mahfuz and K. Ahmed. A review of micro-nano-scale wireless sensor networks for environmental protection: Prospects and challenges. *Science and Technology of Advanced Materials*, 6(3-4), 2005.

[54] F. Martins, L. Lopes, and J. Barros. Towards the safe programming of wireless sensor networks. In *Proceedings of ETAPS 2009*, 2009.

[55] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art (to appear). *ACM Computing Surveys*, 2010.

[56] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *First International Workshop on Data Management for Sensor Networks (DMSN'04), Toronto, Canada*, 2004.

[57] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming Ad-hoc Networks of Mobile and Resource-Constrained Devices. *SIGPLAN Notices*, 40(6):249–260, 2005.

[58] M. Odersky. Programming in Scala. Available at: `http://scala.epfl.ch/docu/files/ProgrammingInScala.pdf`, June 2007.

[59] P. B. Gibbons and B. Karp and Y. Ke and S. Nath and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), October-December 2003.

[60] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000. ACM.

[61] J. Polley, D. Blazakis, J. Mcgee, D. Rusk, and J. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[62] K. Romer and F. Mattern. The Design Space of Wireless Sensor Networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[63] A. Rozinat, M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, and C. J. Fidge. Workflow simulation for operational decision support. *Data Knowl. Eng.*, 68(9):834–850, 2009.

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

[65] D. Simon and C. Cifuentes. The squawk virtual machine: Java on the bare metal. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, New York, NY, USA, 2005. ACM.

[66] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin. PermaSense: Investigating Permafrost with a WSN in the Swiss Alps. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors*. ACM Press, 2007.

[67] I. J. Taylor and B. F. Schutz. Triana - A Quicklook Data Analysis System for Gravitational Wave Detectors. In *Second Workshop on Gravitational Wave Data Analysis*, pages 229–237. Editions Frontières, 1998.

[68] B. Titzer. Avrora: Scalable sensor network simulation. In *In Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN*, pages 477–482, 2005.

[69] W. van der Aalst. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.

[70] W. M. P. van der Aalst, L. Aldred, M. Dumas, and A. H. M. Ter Hofstede. Design and implementation of the yawl system. In *CAiSE'2004*. Springer-Verlag, 2004.

[71] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA . *ACM SIGPLAN NOTICES*, 36(12):20–34, 2001.

[72] D. Vieira and F. Martins. Automatic generation of wsn simulations: From callas applications to visualsense models. In *Proceedings of SENSORCOMM 2010*, 2010.

[73] D. Vieira and F. Martins. Integrating wsn simulation into workflow testing and execution. In *Proceedings of SCUBE'10(to appear)*, 2010.

[74] D. Vieira and F. Martins. Simulação de workflows com integração de simulação de redes de sensores. In *Proceedings of Inforum 2010 (in portuguese) (to appear)*, 2010.

[75] J-SIM website. J-sim website. `http://sites.google.com/site/jsimofficial/`, last access in 03/03/2010.

[76] LabVIEW website. Labview website. `http://www.ni.com/labview/`, last access in 05/09/2010.

[77] QualNet website. Qualnet website. `http://www.scalable-networks.com/products/qualnet/`, last access in 03/03/2010.

[78] Simulink website. Simulink website. `http://www.mathworks.com/products/simulink/`, last access in 05/09/2010.

[79] VHDL website. Vhdl website. `http://www.eda.org/vasg/`, last access in 05/09/2010.

[80] E. Welsh, W. Fish, and J. Frantz. Gnomes: A testbed for low power heterogeneous wireless sensor networks, 2003.

[81] K. Whitehouse, C. Sharp, D. Culler, and E. Brewer. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, 2004.

[82] Wibree. Wibree. `http://www.wibree.com`.

[83] A. Wissner-Gross and D. Alexander. Dielectrophoretic reconfiguration of nanowire interconnects. *Nanotechnology*, 17(19):4986–4990, October 2006.

[84] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, August 2008.

[85] E. Yoneki and J. Bacon. A survey of wireless sensor network technologies: Research trends and middleware's role. Technical Report UCAM-CL-TR646, University of Cambridge, 2005.

[86] J. Zeng and A. Jamalipour. *Wireless Sensor Networks: A Networking Perspective*. WILEY, 2010.

[87] F.L. Zucatto, C.A. Biscassi, F. Monsignore, F. Fidelix, S. Coutinho, and M.L. Rocha. Zigbee for building control wireless sensor networks. In *Microwave and Optoelectronics Conference, 2007. IMOC 2007. SBMO/IEEE MTT-S International*, pages 511–515, November 2007.