Synchronous Macro-Programming of Energy-Efficient Wireless Sensor-Actuator Networks

Marcin Karpiński

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

July 2010

Thesis 9323

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work. I agree that Trinity College Library may lend or copy this thesis upon request.

[redacted]

Kochanej Małgorzacie i moim Rodzicom

Acknowledgements

I would like to thank my supervisor Prof. Vinny Cahill for his support throughout the time I worked in the Distributed Systems Group. I would like to express my deep gratitude to my wife Małgorzata for the patience and understanding she showed all those years (five to be precise), and to my parents who never failed in their encouragement for my efforts. Finally, my best regards should go to: As'ad Salkham, Bartek and Ilonka Biskupskis, Serena Fritsch and Jan Sacha who were always there to remind me that there was a world outside; to Anthony Harrington for being a great desk neighbour; to the Nembes/Emmon team for their patience during my lengthy presentation series; and last but not least, to the entire Distributed Systems Group for being an exceptionally lively environment that I enjoyed spending time in.

Marcin Karpiński
University of Dublin, Trinity College
July 2010

Abstract

Application development for wireless sensor-actuator networks (WSANs) is widely regarded as being a complex task: not only does the programmer need to orchestrate communication and data processing in a heterogeneous network of potentially resource-limited and unreliable devices, but he also needs to ensure timely operation of the system so that the temporal relationship between sensor readings and actuator stimuli is maintained. Macro-programming, originally proposed for the more general domain of wireless sensor networks (WSNs), is a programming paradigm in which applications are concisely specified using concepts whose semantics involve potentially large collections of network nodes, thus, significantly reducing the programming burden. However, macro-programming WSAN applications remains a largely unaddressed problem because the languages proposed to date have either focused on data transformations in the network while neglecting the temporal aspect of WSAN applications, or they do not offer any support for energy-efficient operation.

The synchronous programming programming paradigm was introduced in the 1980's as a way to simplify the development of reactive and real-time systems. The paradigm offers a deterministic concurrency model in which all system components execute and communicate in synchrony according to discrete, global time. As a result, synchronous programs lend themselves to formal analysis and verification, as well as, allow establishing run-time reaction times at compile time. These properties made the synchronous programming languages hugely successful in many control and safety-critical system applications such as, for example, avionics or nuclear power plant control.

This thesis proposes to blend the synchronous programming paradigm with macro-programming for the development of WSAN applications. The synchronous macro-programming model bounds the extent of network communications and imposes a global time regime on the operation of the whole network. We argue that this approach results in the ability to statically schedule all communications with the benefit of network energy conservation, as well as, providing predictable system response times. Furthermore, the global time regime enables network-wide synchronization of sensor sampling and actuation, which is an important property in many control-oriented WSAN applications.

The feasibility of this approach is demonstrated by means of the design and implementation of a synchronous, stream-based programming language - SOSNA. SOSNA is a macro-programming language because a single program defines the operation of a potentially heterogeneous network as a whole, rather than specifying the operation of individual nodes in terms of sending individual messages. The language implements the synchronous paradigm through network time synchronization and through scheduling of all network communications as sequences of network neighbourhood data exchange rounds. Heterogeneity is directly supported via the concept of hierarchically-organised abstract device classes, which are used by the compiler to automatically partition the global program into a number of sub-programs, each of which is to be executed on devices belonging to a particular class.

We have investigated the timeliness, energy-efficiency and reliability properties of SOSNA applications in a number experiments on a network of TelosB motes. Furthermore, several proof-of-concept applications running on a heterogeneous network of TelosB and MicaZ motes, as well as, on a custom-built simulator demonstrate key language features and show that synchronous WSAN programming is a practical concept.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Wireless sensor-actuator[1] networks (WSANs) are an emerging class of networked embedded systems in which the task of controlling a physical phenomenon is realised with a number of spatially distributed sensors and actuators that are interconnected over wireless communication channels. Although WSANs share some characteristics of the more established wireless sensor networks (WSNs) such as, for example, potentially scarce resources of network nodes, they differ significantly from WSNs in that real time becomes an important aspect of application design.

We address the problem of programming energy-efficient and timely WSAN applications by defining a synchronous application execution model in which all network communications are scheduled off-line and comprise a bounded number of fixed-duration network neighbourhood data aggregation rounds. The model rests on the central requirement that time in the network is synchronized and, when used as a basis of a macro-programming language, the model enables the compiler to establish a global application communication schedule at compile time, as well as, to generate application code capable of duty cycling and time-synchronized sensing and actuation. Macro-programming is a programming paradigm that allows to concisely specify network behaviour in terms of constructs whose semantics may involve large collections of network nodes, thus simplifying the programming task. The high-level nature of the macro-programming constructs grants the compiler considerable freedom as to how they are realised in wireless networks and in this thesis we investigate a synchronous approach to macro-programming WSAN applications.

We demonstrate the applicability of the synchronous macro-programming model through the design and implementation of the stream-based, synchronous macro-programming language SOSNA. The language allows to concisely specify a diverse class of heterogeneous WSN/WSAN applications using

[1]Some authors use the term *actor* to refer to the same concept.

a range of generic constructs commonly found in macro-programming languages such as region-based computations or tree and neighbourhood-based data aggregation. The language provides support for expressing feedback, which is an important property of many control-oriented applications, as well as, it transparently handles application heterogeneity by automatically inferring the assignment of different fragments of the global macro-program to network devices of different classes. We evaluate the energy-efficiency and timeliness properties of SOSNA programs in a series of experiments running on wireless networks of TelosB and MicaZ motes (Crossbow Technology 2009).

The rest of the chapter is organised as follows. Section 1.1 describes the core characteristics of wireless sensor-actuator networks and it discusses the challenges related to the development of WSAN applications. Section 1.2 discusses the use of time synchronization in both WSNs and WSANs in order to motivate the central requirement of our model. Sections 1.3 and 1.4 introduce the two corner stones of the model, namely the macro-programming and the synchronous programming paradigms. We conclude the chapter with the presentation of the main contributions of the thesis and of the road map for the remaining chapters.

1.1 Wireless Sensor Actuator Networks

WSANs are envisioned to be used in a wide variety of application domains providing control in potentially large geographical areas. In the field of industrial process control, WSANs are seen as a way of reducing cost by eliminating increasingly complex sensor-actuator cabling. To this end, Song et al. (2008) develop the WirelessHART protocol for industrial control applications that require real-time wireless communications. In agriculture, Sikka et al. (2006) propose to use WSANs in order to enable automatic control of cattle grazing on large fields with the aim of reducing the amount of effort required to maintain the fields, while Stipanicev & Marasovic (2003) investigate the suitability of WSANS for greenhouse climate control. In the field of building management systems, example applications range from structural control systems for earthquake response (Lynch et al. 2007) to automatic control of lighting and air-conditioning systems (Li 2006, Lin et al. 2002).

The CSOnet system (Montestruque & Lemmon 2008) is one of the best documented examples to date of a WSAN implementing distributed control over a large spatial area. The system is a metropolitan-scale WSAN that exercises control over a city's sewage system in order to minimise the frequency of overflows. Combined sewer overflow (CSO) is an event during which storm waters add up to sanitary flows and cause an overflow, which is typically directed to a nearby river. Naturally, CSO events pose an environmental threat and their frequency should be minimised. Also, since the federal law in the United States fines municipalities for each sewer overflow, there is a incentive to the development of a system that helps to reduce the annual cost of running a city's sewage system.

CSOnet attempts to reduce the frequency of CSO events by allowing to temporally store the excess storm waters within the sewage system itself. Such an approach requires distributed real-time monitoring and control of waste water levels so that they are released to the river only when their levels exceed the sewer system's capacity. To this end, CSOnet employs four types of devices: water level sensor nodes (INodes), communication relay nodes (RNodes), gateway nodes (GNodes) and actuator nodes (ANodes). The gateway nodes are responsible for providing connectivity to a wide area network, while the actuator nodes control waste water diversion structures directing the incoming water either to a waste water treatment plant or directly to the river.

In order to enable scalability, the system adopts a hierarchical architecture in which the global, metropolitan-scale network comprises a federation of smaller WSANs that are interconnected over an existing wide area network. Each of the individual WSANs controls a section of the sewage system and comprises multiple INodes, one ANode, at least one GNode and, depending on communication conditions, a number of RNodes. The INodes measure water levels upstream from their corresponding ANodes and the RNodes are used to connect SNodes and ANodes with the nearest gateway, in the case when there is no line-of-sight among the devices. In order to implement the distributed control law governing the use of the actuators, CSOnet follows a particular communication pattern, in which data from all INodes are routed to their corresponding gateways, which then exchange it with each other, compute the global control law and send appropriate commands to the actuators.

The control law implements a feedback-based optimal control strategy that minimises the amount of sewer discharged to the river. The strategy switches between two modes of which the first is used by those ANodes that have enough capacity to direct all of the incoming water to the waste water treatment plant, while the second mode is used by the remaining ANodes to compute the minimal amount of waste water to be directed to the river. The control law is scalable, i.e., the actuator nodes can make their decisions on their own once they know the current values of few global parameters the computation of which is simple (it involves summation of distributed values), and the value of one local parameter that is computed based on the information from neighbouring actuators.

CSOnet implements time synchronization across the entire network in order to both synchronize sensing and actuation, as well as to implement duty cycling for network energy conservation. A common time base for sensors and actuators is necessary in order to ensure correct execution of the control law, which is specified in the language of optimal control theory (see, for example, Athans & Falb 2006). On the other hand, since both the INodes and RNodes are relatively small battery-powered devices, duty cycling their activities becomes vital to the realistic applicability of the whole system. As far as control is concerned, the required levels of time synchronization accuracy are closely related to the dynamics of the controlled physical phenomenon, which in the case of waste water level changes are not stringent. The implementation of network duty cycling, however, poses additional requirements since misalignment in sleep periods across the network might lead to broken communication links and, thus, it might disrupt the operation of the whole system. Also, since CSOnet resynchronizes the network only once a day in order to conserve energy, it is expected that time synchronization be precise. As a result, the nodes are able operate at two per cent duty cycle, i.e., they stay dormant 98% of time.

When network nodes are in the active mode, they engage in communications. Because the network operates in harsh environment of city sewer system, extra measures are taken in order to improve communication reliability. CSOnet takes a best effort approach that uses explicit packet acknowledges and retransmissions, which are performed maximally until the end of each active mode time slot. As a result, the system achieves 99% of end-to-end data yield and the small fraction of lost messages is tolerated by the control algorithm.

CSOnet is an example of a heterogeneous WSAN. Such networks comprise different types of devices that may range from various kinds of sensor and actuator nodes, through data processing nodes (controllers) to network repeaters and gateways. Homogeneous WSANs, on the other hand, comprise nodes with both sensing and actuation capabilities such as, for example, robot swarms (Oung et al. 2010). Despite this variety, however, it is typically assumed that WSANs, among other types of nodes, comprise low-cost devices with scarce energy budgets, relatively short range and unreliable radio communications (Akyildiz & Kasimoglu 2004, Verdone et al. 2008).

WSANs share many similarities with wireless sensor networks. Multi-hop network topologies might be used as a result of the fact that long-range communications require a reliable communication medium and are energy consuming. Data from multiple sensor nodes might need to be fused in order to obtain reliable information because of the low-cost nature of sensor nodes. The underlying need for energy-efficient operation, as well as, the bandwidth limitations of wireless channels result in more emphasis being put on in-network data processing instead of simply communicating sensor readings to data processing nodes. This fact is further motivated by the work of Pottie & Kaiser (2000) who showed that radio hardware might consume orders of magnitude more energy than the processor. Also, Raghunathan et al. (2002) discovered that there was no significant difference in energy consumption between a radio in the transmission and idle states. Hence, designing communication protocols that make extensive use of radio duty cycling is vital to achieving network lifetimes in the order of years.

For example, Burri et al. (2007) propose a data gathering protocol for wireless sensor networks that integrates TDMA-style medium access protocol with a tree network topology so that duty cycles of the order of 0.2% are achieved, extending their network's lifetime up to approximately five years.

There are, however, important differences between WSANs and WSNs. The addition of actuators to the network closes the loop of information flow in the system and introduces some real-time bounds on network operation as the system is now expected to react to changes in the state of the environment (Akyildiz & Kasimoglu 2004, Zhao & Guibas 2004). Such a view makes WSANs essentially a class of distributed control systems, one in which numerous sensing, control and actuation components are spatially distributed and communicate over potentially unreliable links. Akyildiz & Kasimoglu (2004) point out that control, being the main application goal, affects the range of suitable WSAN topologies making large multi-hop networks less useful since it is more difficult to provide them with communication timeliness guarantees. This view is supported by Arora et al. (2005) who, based on the experience gained during the deployment of a large sensor network, note that reliability of multi-hop communication spanning more than five or six hops may be insufficient for practical use.

WSAN application development is widely considered to be a very difficult and complex task. At the high level, the application could be viewed as a distributed control system that has to dynamically react to a changing environment in such way that the application's goals are achieved. Formal control design methodology[2] might need to be used at this stage as in, for example, Wan & Lemmon (2007) and Lynch et al. (2007). Then, a system architecture needs to be designed in such way that various functional and non-functional application requirements are satisfied. This includes choosing a network topology capable of providing the desired communication quality and timeliness levels. The control laws and logic need to be translated into a distributed coordination protocol that will orchestrate processing of information flowing from sensors to controllers and further towards actuators in a timely and, most importantly, energy-efficient manner. For example, Melodia et al. (2005) developed a dynamic cluster-based protocol that addresses both the timeliness and energy efficiency aspects of WSAN coordination. Additionally, since different hardware could be used for sensors, actuators and controllers, some of which having potentially severely constrained resources, the programming task might involve writing a number of different, but inter-communicating programs for a number of different hardware platforms.

Domain-specific programming languages can facilitate the development of WSANs through automatic handling of certain aspects of their operation. Based on the above discussion, we conclude

[2]Unfortunately, there is no universal set of techniques that can be applied, especially in the WSAN domain. Sinopoli et al. (2003) discuss the suitability of existing models of computation for the WSAN domain while D'Andrea & Dullerud (2003) report initial results on methods for control of interconnected systems.

that the following characteristics are common to many WSANs and hence should be supported by programming languages that target this domain.

- **Network heterogeneity.** Unified handling of different hardware and software platforms within a single overall framework can greatly simplify the development task. This includes also the ability to handle in an uniform manner the communication among devices of different kinds.

- **Stringent resource constraints.** It is envisioned that many WSANs will comprise devices with highly constrained resources. Therefore, programming tools should be able to produce application code that is efficient in terms of its time and space requirements. Also, since the many WSAN nodes are likely to be battery powered, support should be provided for automatic duty cycling of node's hardware components including, in particular, the radio transceiver.

- **Timely operation.** The ability to establish bounds on program reaction times is of fundamental importance since the main task of WSANs is to interact with the surrounding physical environment. Because WSANs are distributed systems, their operation involves communication and, hence, system reaction times necessarily include delays introduced by communication protocols. Additionally, since many control algorithms rely on the ability to determine the time when sensors are sampled and when actuators are instructed, support is necessary for synchronization of the sensing and actuation tasks across the network.

- **Data aggregation.** One of the main advantages of WSANs (as well as WSNs) is the ability to use multiple sensors spread across an area. Therefore, support is necessary for fusion of data coming from multiple network nodes.

- **Data propagation.** The need to actuate based on the values of sensor readings translates to the need to communicate control decisions to the actuators, thus, making the established WSN base station-centric communication models insufficient for many WSAN applications.

- **Feedback.** Many control techniques combine information on past system states with the current state in order to improve the accuracy of control decisions. It is necessary, therefore, that WSAN programming models allow references to past program states, particularly in scenarios when the state may be distributed across different network nodes.

1.2 Time synchronization in WSANs

Because network time synchronization is at the core of the synchronous macro-programming model, its feasibility for WSAN applications underlines the practicality of our approach. There is a large body of literature in the WSN domain on time synchronization protocols that take into consideration such WSN characteristics as limited hardware resources, tight energy budgets, unreliable communication or low-quality hardware clocks (see Römer et al. 2005 for an extensive survey). Many of these methods can be directly applied to WSANs, and even though there could potentially be space for further research exploiting, for example, the heterogeneous nature or some network topologies typical to WSAN systems, we are not aware of any significant work taking this approach. Hence, in the remainder of the thesis, we do not make a distinction between time synchronization methods for these two domains.

There are a number of important applications of time synchronization wireless sensor networks. Energy-efficient network operation can be enabled if nodes are able to agree on their active and sleep operation periods. Such approach is taken, for example, by Hohlt et al. (2004) who propose a decentralised protocol for coarse-grained synchronization of communication over a tree topology. Madden et al. (2005) use a similar technique to schedule communications and sleep times of sensor nodes that process data queries. Time-synchronized sensor sampling is also required in applications performing distributed estimation tasks such as, for example, the VigilNet target tracking system (He et al. 2006), the counter-sniper system of Simon et al. (2004) or the volcano eruption monitoring system of Warner-Allen et al. (2005).

WSAN applications introduce additional requirements for time synchronization. Akyildiz & Kasimoglu (2004) note that, apart from the need to base control decisions on sensor data sampled by different sensor nodes at approximately the same time, actuation may need to be exercised at some precisely chosen moment in order to achieve a given control goal. For example, Zhao et al. (2007) discuss an application that requires tight synchronization of a number of cameras operating at a football field so that smooth image transitions could be composed. Lynch et al. (2007) and Montestruque & Lemmon (2008) need time-synchronized actuation so that their theoretically derived control laws could be correctly implemented on a real network, while Dalton et al. (2008) recognise this issue by providing explicit support for synchronized actuation in their programming language DESAL, which was designed with WSAN applications in mind.

Time constraints on sensing and actuation in WSAN applications lead to constraints on communication and processing, which are typically performed after the sensors are sampled and before the actuators can stimulated. As advocated by Stankovic et al. (2003), timely communication and actuation are some of the most fundamental requirements of WSAN applications because they operate in close relation with the physical environment in which they are embedded. As a result of this fact, a number of real-time communication protocols have been proposed for the domain. For example, the Speed protocol (He et al. 2003) delivers soft real-time communication guarantees by maintaining a desired data delivery speed through the use of feedback control of a geographic routing protocol. Rowe et al. (2008) developed a reliable and energy-efficient TDMA communication scheme that makes use of custom-built hardware, in order to increase the accuracy and efficiency of global network time synchronization. A more fundamental approach is taken by Prabh (2007) who analysed the real-time capacity of wireless sensor networks and developed a number of communication protocols that demonstrate the predicted timeliness levels.

We conclude that time synchronization is feasible in WSANs. The synchronous macro-programming model, however, does not specify any particular network time synchronization scheme because different applications might have different requirements with respect to both the desired accuracy and the cost of time synchronization. We leave it, therefore, up to the implementation of the model to choose the time synchronization method and we require only that this method can be made transparent to the semantics of the model, e.g., its operation can be included in the global communication schedule.

1.3 Macro-programming

Macro-programming (Gummadi et al. 2005) is a relatively novel programming paradigm proposed as a way to radically simplify the development of distributed applications comprising many, wirelessly-communicating nodes by abstracting away most of the complexities involved in orchestrating communication among them. Following the taxonomy in Sugihara & Gupta (2008), WSN programming models can be roughly classified into three main groups:

- *node-level* models that focus on specification of the behaviour of individual nodes,
- *group-level* models that offer abstractions that are related to groups of network nodes, for example, n-hop network neighbourhoods or target-tracking groups,
- *network-level* models that operate on concepts that may potentially span across the entire network.

Macro-programming languages fall into the last category and offer an unprecedented level of abstraction since fully-functional WSN applications can be written in as few as tens of lines of code, compared to hundreds, when node-level or even group-level programming models are used. This property has been consistently reported by different authors - see, for example, Whitehouse et al. (2004), Luo et al. (2006) and Newton & Welsh (2004). The main idea behind macro-programming is to be able to address the operation of the whole network using programming constructs whose semantics spans across multiple network nodes. As a result, the compiler gains significant freedom as to how program execution proceeds within the network enabling, on one hand, optimisations that are out of reach for compilers of lower-level languages, while on the other, significantly adding up to the complexity of developing the compiler. Additionally, being high-level, macro-programming languages may introduce language expressiveness limitations when compared to lower-level programming languages simply because it is the compiler that arranges the details of network communication and not the programmer.

There is a considerable variety in the range of abstractions proposed by the existing macro-programming languages and it is often motivated by a particular class of applications that the languages target. Some of the earliest works impose a database abstraction on the network and use a subset of the SQL language in order to dynamically query network nodes for their current sensor readings. Others take approaches inspired by functional programming languages treating the whole network as a single distributed computer, or by imperative programming languages adding programming constructs that allow for convenient specification of distributed computations.

1.4 Synchronous Programming

The synchronous programming paradigm (Benveniste et al. 2003) was introduced in the 1980's as a way of formalising the development of safety-critical real-time systems. The paradigm is founded on the synchronous hypothesis that states that program reactions to signals or events are instantaneous, i.e., they have a negligible duration. Synchronous programs, therefore, are driven by logical time that is defined as a sequence of system reactions to input events. Thanks to their deterministic model of concurrency and well defined semantics, the synchronous languages are well suited for formal verification, as well as, they enable calculation of program response times and memory use at compile-time. This property made the approach hugely successful in such application domains as automotive, avionics, aerospace or nuclear power plant control where formal verification of safety critical code is mandatory. A number of synchronous languages has been developed over the last twenty years. Some, like Esterel (Berry & Cosserat 1985), Lustre (Caspi et al. 1987) and Signal (LeGuernic et al. 1991) are already industry standards for safety-critical embedded software development. Other, like Lucid Synchrone (Pouzet 2006), are research prototypes for experimentation with novel techniques.

The synchronous programming paradigm is a general concept unrelated to the syntactical details of the existing synchronous languages. As discussed by Benveniste et al. (1999), there are three key requirements for a programming language to be synchronous:

1. Program operation progresses in discrete steps and can be modelled as an infinite sequence of reactions.
2. It is possible that, within a single reaction, programs make decisions based on event *absence*. For example, program can output a default value if one of its inputs is absent.
3. Parallel composition of two sub-programs (e.g., two program modules, threads, streams, etc.) is defined by taking the pairwise conjunction of their individual reaction sequences, whenever the programs are composable according to language's semantics.

Such an abstract formulation led to consideration of distributed implementations of synchronous languages. The globally asynchronous, locally synchronous (GALS) model that describes a network of asynchronously communicating processors that themselves operate synchronously was suggested to be the overall architecture. Benveniste et al. (1999) observed, however, that a reliable communication medium is required so that the synchronous semantics can be preserved: messages cannot be lost and have to be delivered in the same order as they were sent. Caspi et al. (1999) proposed a general algorithm for distributing synchronous programs among a number of processors and later (Caspi et al. 2003) described a compilation technique of Lustre programs to the Time-Triggered Architecture (TTA), which is a distributed, synchronous and fault-tolerant communication bus architecture used in the real-time systems domain (Kopetz 1997). Ongoing work investigates distributing synchronous programs on less reliable architectures such as the loosely time-triggered architecture (LTTA) (Benveniste et al. 2007) or on event-triggered communication architectures (Romberg & Bauer 2004).

1.5 Contribution of the Thesis

This thesis proposes to blend the synchronous programming paradigm in the context of macro-programming wireless sensor-actuator networks. As it was shown in previous sections, time-synchronized network operation is vital to many WSAN applications. Synchronous languages, on the other hand, because of their static and deterministic semantics give the compiler considerable freedom as to how program execution is realised and permit compile-time calculation of such important properties as program response times or memory use. We propose a macro-programming model in which synchronous semantics are given to communication primitives so that, assuming network time synchronization, the operation of the whole network can be globally scheduled by the compiler. As a result, applications execute in a timely and predictable manner and they implement network-wide duty cycling for energy conservation. Also, the synchronized network operation allows for synchronization of sensor sampling and actuation among all network nodes.

The synchronous macro-programming model is centred around the network neighbourhood communication paradigm in which nodes located within each others' radio communication range periodically exchange information. Whitehouse et al. (2004), in their work on the Hood programming abstraction, advocate that the neighbourhood communication paradigm is particularly well suited for WSN/WSAN applications because it is localised, i.e., it does not require any information that is not local to sensor nodes, as well as, because it naturally matches the broadcast nature of wireless communications. In the context of programming languages, neighbourhood-based communication is sometimes used as a means to implement local state sharing, for example, in the DESAL (Dalton et al. 2008) programming language, or as a primitive communication operator, as in the Proto macro-programming language (Beal & Bachrach 2006).

Similarly to Proto, we use the neighbourhood communication primitive as an elementary building block that can be used to construct higher-level communication operators. Observing that WSAN applications must bound the processing time between sensor sampling and actuation, we constrain the duration of the neighbourhood communication primitive to a fixed value and limit its use so that a static communication schedule of known size can be composed at compile time. In this way, network operation proceeds in *cycles* of fixed duration, each of which comprising a fixed number of neighbourhood communication *rounds*. The main consequence of static communication scheduling is that macro-programs are limited in terms of the extent of network communication to a certain number of hops - it is no longer possible to directly exchange information between two arbitrarily distant network nodes. The extent the communication bound, however, is application-specific rather than universal and this limitation is in agreement with what is expected of WSANs: as mentioned earlier, general considerations of the domain's timeliness requirements lead to favouring smaller topologies while the reliability of multi-hop communication over five-six hops was reported to be below that required for practical use.

We demonstrate the feasibility of our model through the design and implementation of the synchronous macro-programming language SOSNA. Inspired by the Lustre and Lucid Synchrone languages, SOSNA offers a functional stream-based approach to programming heterogeneous WSAN applications. The macro-programming aspect of the language follows from the introduction of multi-hop network topologies that guide the aggregation and propagation of distributed stream values, as well as, from the way the language addresses network heterogeneity in which a single program defines the operation of the whole network by means of constructs whose semantics might span devices of different classes. Both aggregation and propagation are treated as primitive operations whose semantics are defined in terms sequences of network neighbourhood communication rounds and the extent of which is a compilation parameter tunable to reflect the application's needs. SOSNA is also a synchronous language in the sense of the definition from Section 1.4, meaning that all program streams are evaluated in lock-step according to the global logical clock. As a result, time-synchronized operation is imposed on the whole network so that timely communication among different network nodes can be realised. Heterogeneity, as a fundamental characteristic of WSAN applications, is explicitly supported in SOSNA by the concept of device classes that describe the different kinds of nodes that comprise the network, and which are hierarchically organised to reflect typical WSAN architectures as described, for example, by Akyildiz & Kasimoglu (2004). Device class information is used to annotate program streams so that the compiler can automatically partition the program into a set of sub-programs that are to be executed by network nodes of the corresponding types.

The contributions of this thesis are the following:

1. We merge the synchronous programming and the macro-programming paradigms for the development of heterogeneous WSAN applications. We show that this approach carries a range of benefits, in particular, it permits to establish bounds on system response times at compile-time, to synchronize sensing and actuation in the network and to automatically generate application code that realises duty cycling for energy conservation.

2. We designed and implemented the synchronous macro-programming language SOSNA, which to the best of our knowledge, is the first macro-programming language to provide the above characteristics. We show that the language can be used to develop a diverse class of WSAN applications, thus strengthening the case for synchronous macro-programming.

3. We show that synchronous macro-programming can be implemented on resource-constrained hardware platforms with measurable benefits. The timeliness and energy efficiency of SOSNA applications are assessed by means of a number of experiments performed on wireless networks of TelosB and MicaZ motes, as well as, memory usage of example applications is discussed based on the results of program compilation for the TelosB and MicaZ targets. We show that the applications execute in a timely manner and that our duty cycling policies result in a significant decrease in battery power consumption. Also, we show that the prototype SOSNA compiler is capable of generating memory-efficient code that fits on resource-constrained devices.

1.6 Thesis Outline

Chapter 2 reviews the existing work in the field of programming languages for wireless sensor and wireless sensor-actuator networks, as well as, it analyses the differences between synchronous macro-programming and the existing methods for automatic distribution of synchronous programs. In Chapter 3, we describe the synchronous macro-programming model and provide a description of the SOSNA programming language. The main features of the language are presented and are followed by a description of the language's semantics and of its type system. Finally, a number of example programs are given in order to demonstrate that the language can be used to develop a diverse class of WSAN applications. Chapter 4 discusses the implementation of the SOSNA compiler. The main program transformation stages are described, including type inference, scheduling and program partitioning. The second part of the chapter presents the NesC compilation target and it describes the overall framework of program execution in wireless network of TinyOS-supported devices. We evaluate the key characteristics SOSNA applications compiled to NesC target in Chapter 5 by presenting the results of a series of experiments performed on wireless networks of TelosB and MicaZ motes. The timeliness and energy-efficiency properties are quantified and the chapter ends with the analysis memory requirements of the compiled applications. We conclude this dissertation with a discussion of the limitations of the existing SOSNA implementation, as well as, of directions of its improvement and further research in Chapter 6.

Chapter 2

Related Work

This chapter reviews the existing work in the field of programming languages and frameworks for the domains of wireless sensor and wireless sensor-actuator networks. The chapter also presents the most prominent synchronous languages and it discusses the relationships between synchronous macro-programming and the techniques for distributed execution of synchronous programs. A particular emphasis is put on macro-programming languages since these are the most related to our work, as well as, because a detailed review of all existing programming models for the WSN/WSAN domain is beyond the scope of this thesis. Since much of the work covered by this chapter is aimed at wireless sensor network applications we summarise it in terms of its applicability to the WSAN domain and reflect on the similarities to the work presented in this thesis.

2.1 Wireless Sensor/Actuator Networks

This section discusses the existing programming languages and frameworks for both the WSN and WSAN application domains. The structure of this presentation follows the general classification of WSN programming models proposed by Sugihara & Gupta (2008) and introduced in Section 1.3 which groups all WSN programming models into three broad categories: network-level, group-level and node-level programming models. The authors partition the network-level programming model category into two subcategories corresponding to macro-programming languages and database abstractions respectively. We make a small departure from this categorisation and discuss macro-programming languages and all other network-level programming models in separate sections. Then, we present group-level programming models and we conclude the section with an overview of node-level programming models.

2.1.1 Macro-programming Languages

This section discusses the suitability of the existing macro-programming languages for WSAN application development. We focus on four properties which are important to wireless sensor-actuator networks applications and which were motivated in Chapter 1: language support for timely execution, including the support for time synchronised sensing and actuation, language support for energy-efficient operation, network heterogeneity and the ability to express feedback-driven behaviour. We group the existing macro-programming languages into three general categories which correspond to the common characteristics of the languages. Because stream-based macro-programming abstractions appear to be the most commonly proposed, as well as, because the SOSNA programming language described in this thesis is also a stream-based formalism, we focus our attention on this group.

2.1.1.1 Stream Processing

Regiment (Newton et al. 2007) is a functional macro-programming language which was inspired by the idea of functional reactive programming (FRP) originally proposed by Elliott & Hudak (1997). In Regiment, the programmer views the network as a collection of spatially-distributed and time-varying signals which may correspond, for example, to sensor readings or to results of computations based on them. The concept of regions is used to group spatial collections of signals or other regions with the purpose of transforming them. Regiment offers a range of built-in region constructors which include spanning tree-based regions such as the k-hop network neighbourhood of a selected sensor node, a circle, i.e., a group of nodes located within a given physical distance from a node of choice, or regions formed through network neighbourhood information propagation. Regions can be transformed with the use of three generic region operations: function application to values held by region members, filtering out region's members and folding all values comprising the region to a single value through iterative application of a user-specified function. This is similar to how SOSNA handles network data transformations and we describe it in more detail in Chapter 3. Regiment programs are transformed to static, asynchronous dataflow graphs of region operators and are compiled down to the Token Machine Language (Newton et al. 2005) after the application of an extensive set of optimisations. TML defines an event based communication model and has been developed to serve as an intermediate representation of high-level macro-programs. Regiment takes a base station-centric approach, i.e., the final result of a networked computation is expected to be routed towards a base station.

Wavescript is a stream processing language developed by Newton et al. (2008) and designed for applications which deal with high-volume, asynchronous data streams. Typically, high-rate stream processing systems report performance issues since time stamping of individual data values incurs additional cost. Wavescript improves on that by introducing the concept of signal segments which are used to group stream values in blocks of fixed size allowing, this way, the compiler to better optimise the code. The language does not provide any support for in-network data transformations but rather targets general-purpose stream processing applications. Nevertheless, thanks to automatic program distribution techniques developed by Newton et al. (2009), Wavescript can be used to build base station-centric WSN applications without the need to address the networked aspect of application's operation.

Flask (Mainland et al. 2008) is another FRP-inspired programming language which targets resource-constrained stream processing WSN applications. Flask is a domain specific embedded language (DSEL) in the sense of the formulation given by Hudak (1998), i.e., it is embedded in a general-purpose functional language - Haskell in this case. The main focus of this work is to investigate novel translation and compilation techniques for the generation of efficient code on resource-constrained platforms. The language focuses on processing of asynchronous data streams but, in contrast Wavescript, it provides a generic neighbourhood communication primitive which can be used to implement higher-level networking primitives such as, for example, network-wide spanning tree-based data aggregation.

A biologically-inspired approach to macro-programming is proposed by (Beal & Sussman 2005). As a result of their work on the Amorphous Computing project, the authors developed Proto - a functional stream-based language for programming networks of potentially large number of highly resource constrained and locally communicating devices. Having originally thought of such applications as the Paintable Computer (Butera 2002), the authors extended the scope of the work and implemented a compiler for the WSN domain (Beal & Bachrach 2006). Proto is similar to Regiment in the sense, that programs are defined as transformations of spatial regions of time varying values. The regions, however, are not part of the type system hence all transformations other than point-wise function application have to be explicitly implemented by the programmer using the universal neighbourhood communication operator. Multi-hop communication is realised via feedback, i.e., through recursive use of the neighbourhood data exchange operator. In order to keep the program semantics general, Proto introduces the notion of amorphous medium abstraction which represents an idealised, continuous computational substrate instantiated at runtime by a concrete network topology. The language provides explicit support for actuation which cannot be easily synchronized since application execution is only partially synchronous, i.e., it proceeds in rounds of equal duration while time among network nodes is not synchronized.

Another stream processing system is presented by Awan et al. (2007). COSMOS is a macro-programming framework which targets heterogeneous WSAN applications and which comprises the mPL programming language and the mOS portable operating system used for running mPL programs on different hardware platforms. COSMOS offers a static dataflow programming model, similar in essence to that of Wavescript, in which applications are composed as static graphs of stream processing components (the authors use the term *functional components*), windowing is used to pack multiple data items into a single block for optimisation reasons, and queueing is used to share data items among different components. In addition, the framework allows partitioning of the dataflow graph for networked execution. COSMOS differs from Wavescript primarily in linguistic terms: the language is not functional, instead the processing components are programmed in a subset of C and the mPL language is used to explicitly define component wiring and partitioning. Contrary to Wavescript, COSMOS targets a tiered system architecture in which heterogeneous devices of potentially different capabilities are hierarchically organised into a tree network topology. The language allows data to flow in both directions over the global tree topology thus giving way to the implementation of feedback which is an important component of many control systems. Additionally, support for in-network data processing is provided by a k-hop network neighbourhood communication component and the language is capable of expressing actuation since the processing components are programmed using a subset of C.

ATaG (Bakshi et al. 2005) is a visual macro-programming formalism which implements a static dataflow programming model similar to that of COSMOS. The processing nodes (*tasks* in authors' terminology) are specified using an imperative language such as, for example, Java. The concept of abstract data items (which roughly correspond to stream output queues in COSMOS) is used as means of information exchange among tasks which might potentially reside on different network nodes. Abstract communication channels are introduced for the creation of new or reading the values of the existing data items. The channels may be specified to connect tasks residing on the same network node or they may be annotated to share data items among groups of nodes. The set of different sharing groups that can be expressed in the language defines the language's communication model. ATaG allows sharing groups to be defined that range from k-hop network neighbourhoods, through network clusters, parent or children nodes within the global tree topology, to the set of all nodes in the network. Since the tasks can be placed on arbitrary network nodes, as well as, because they are specified in an imperative language, networked actuation can be expressed. Hence, ATaG can be used to develop applications for the WSAN domain.

Semantic Streams (Whitehouse et al. 2008) is a declarative sensor network query system designed with the goal of providing composable inference over data streams coming from heterogeneous sensors.

The system is based on logic programming (Tamir & Kandel 1995) and it allows to prove queries from sets of facts and relations defined over them. Semantic Streams assumes that a range of inference units are available to the programmer, each of which being an opaque component realising some elementary inference task like, for example, vehicle detection based on raw sensor data. The user can pose high-level queries such as, for example, the ratio of cars to trucks in an underground parking system, without having to worry which sensors to use, how to combine data from different sources or how to interpret raw sensor readings. Constraints can be introduced in order to specify non-functional aspects of the query. For example, a detection confidence threshold can be given in cases when there are more than one answers to a query due to, for instance, the existence of several different sensors with overlapping detection zones. Semantic Streams aims to be used for infrastructure-based sensor networks such as, for example, networks of climate sensors in buildings or presence detectors in parkings or streets because these networks usually are not energy-limited and there is no need for in-network processing since all sensor nodes are likely to be connected to the central server. As a result, the system is not well suited to the development of WSAN applications other than following a centralised control pattern.

Stream processing has been shown to be a good match for applications whose operation is driven by time-varying sensor data. Even though WSANs fit in this general description, the languages presented above differ in their capability to express a wide range of WSAN applications. The most fundamental requirement for a WSAN programming language is the ability to express networked actuation, i.e., to be able to select and instruct, given the results of sensor data transformations, a suitable set of actuator nodes. Regiment, Wavescript and Flask are purely functional languages and take the base station-centric approach. Hence, the only way to realise actuation of some sort is to connect the output of the global computation to a centrally controlled actuator. Although such an approach would definitely find its use, it is not flexible enough for all conceivable WSAN applications.

Feedback is a general technique of using past system outputs in order to influence its current or future outputs and is a basic principle that drives the design of many control systems. It is desirable, therefore, that WSAN programming languages be able to express it. Regiment is the only macro-programming language which is lacking means to either reference past stream values or to define some form of persistent state variables which could be used to store past stream values. Wavescript allows persistent variables to be explicitly defined while the other languages provide constructs to loop their dataflow graph definitions.

As explained in Chapter 1, timely sensing and actuation are important characteristics of many WSAN applications. All of the presented stream processing languages can implement synchronized sensing by means of extending the run-time system with a time synchronization service because all these languages assume a periodic execution model in which application's operation is driven by a local, periodic timer. Actuator synchronization, however, is more difficult of a problem since the duration of data processing and communication tasks must be known or be bounded. In fact, none of the discussed languages offers any guarantees or means to schedule timely actuation.

Energy-efficiency is another important issue in the WSAN domain. We noted in Chapter 1 that keeping low communication overhead is not sufficient to conserve energy in resource-limited sensor nodes because the radios when left in the idle (listening) state consume almost the same amount of energy as during sending or reception of messages. Unfortunately, none of the presented languages provides support for duty-cycled operation. Moreover, because all of these languages, apart from Proto, target processing of asynchronous data streams it is difficult to see how this mode of operation could be incorporated. Proto, on the other hand, is based on a partially-synchronous execution model hence it could be possible in theory, assuming network time synchronization, to construct a global duty cycle. However, because of the relatively low-level nature of the language's only communication primitive - the neighbourhood data aggregation operator, Proto imposes a programming model in which data values need to be forwarded for unbounded number of hops, thus making one round of execution a time unit of relatively fine granularity. As a result, the introduction of automatic duty cycling at the level of one round would result in large application response times. Nevertheless, such a time-synchronized and round-based execution model is the basis of the work presented in this dissertation. Contrary to Proto, our model is based on a coarser-grained time unit of several neighbourhood aggregation rounds, the number of which is application-specific and established at compile-time.

Extensive support for network heterogeneity among the presented macro-programming languages is provided only by COSMOS and ATaG. The former proposes a hierarchical model in which more powerful devices reside closer to the global tree network topology what is in agreement with WSAN topologies suggested in the literature (see Chapter 1 for an overview). The latter does not pose any limitations on the assignment of tasks to different hardware nodes. It is not clear, however, what the real implementation of the language would be like since the existing publications fail to present this matter with sufficient degree of detail. From among the remaining languages, only Wavescript provides a limited support for application execution on heterogeneous hardware. In this model, a base station is expected to do the bulk of application stream processing tasks and initial processing of sensor data may be delegated to a network of homogeneous sensor nodes.

2.1.1.2 Imperative Languages

This section reviews macro-programming languages that follow the imperative programming paradigm. One of the key characteristics of this paradigm is that persistent variables are used to store application state and computation progresses through the updates of the variables' values. This is in contrast to functional stream processing in which data values flow through a network of stream operators and they need not be allocated persistent storage. As a result, distributed state synchronisation is not required enabling, therefore, simpler distributed implementations. On the positive side, the imperative programming paradigm makes actuation easy to express and, since it is well established, it might be favoured by some programmers.

Macrolab (Hnat et al. 2008) is a macro-programming language built around the idea of distributed vectors. Borrowing its syntax from Matlab (a standard tool among scientists and engineers), the language offers a programming model in which data values stored at different network nodes are collectively represented as *macrovectors*. Each macrovector corresponds to a spatial collection of values which are addressed by network identifiers of the nodes which hold them. Macrovectors can be filtered, indexed and modified in a similar way to Matlab's vectors and matrices. One of the language's key features is the fact that the compiler can automatically choose how macrovectors are represented in the network based on a target deployment network model. The authors describe the three currently supported macrovector representations. Centralised macrovectors are stored entirely on on the base station and updates to their composite values are transmitted by the corresponding network nodes. Fully decentralised representations store composite values on the corresponding network nodes and communication is required for macrovector operations involving fusion of different composite values (for example, finding macrovector's maximum value). Finally, fully duplicated representations assume storing the whole macrovector at every network node and all updates to macrovector's components need to be globally communicated. Macrolab explicitly targets the WSAN application domain[1], hence actuation, feedback and synchronized sensor sampling (assuming a time synchronization service is included in the runtime system) can be easily realised in the language. Energy-efficient operation, however, is only partially supported since the compiler optimises application cost by focusing solely on minimisation of the number of messages sent and does not provide for duty-cycled network operation. Also, it is not clear whether synchronized actuation can be performed since programs can specify only periodic operation while the automatic selection of macrovector representation makes application communication times vary for different deployment topologies. Finally, heterogeneity is not directly

[1]The authors actually use the term Cyber-Physical Systems (CPS) but we see this as equivalent to what is referred to in the literature as WSANs.

supported by the language since the compiler distinguishes only between the base station and the sensor nodes.

A sequential model of programming distributed wireless sensor network applications is investigated by Kothari et al. (2007). The Pleiades language has a C-like syntax and uses a special-purpose looping operator *cfor* to iterate over actions performed by different network nodes. Program execution in the network allows a degree of parallelism but the language's semantics guarantee that its computational effect will correspond to some sequential ordering of network activities providing this way strong consistency guarantees. Locking, synchronization and deadlock avoidance mechanisms are used to enable application control flow migration among different network nodes. Being an imperative language, Pleiades can easily express feedback, actuation and time-synchronized sensor sampling. However, even though the language was used to develop applications which require time synchronized operation like, for example, the pursuer-evader game in which WSN tracks an evading target and reports its position to autonomous pursuers, the overhead of application state synchronization significantly affected application response times (Kothari et al. 2007). Moreover, since the programming model does not address the temporal dimension of application execution, neither the compiler nor the runtime system can make decisions on the duration of program execution times. This makes both duty cycling and time-synchronized actuation difficult to achieve. In fact, the authors confine themselves only to the discussion of the messaging cost of their macro-programs.

A similar programming model is offered by the Kairos (Gummadi et al. 2005) language which embodies work that preceded the development of Pleiades. Contrary to its successor, Kairos does not give strong execution consistency guarantees, instead, the language provides means to define application-specific fault recovery mechanisms (Gummadi et al. 2007). As far as WSAN application development is concerned, both Kairos and Pleiades offer a similar degree of support.

The SpatialViews (Ni et al. 2004) programming language exploits a similar approach to programming distributed computation in which sequential iteration over a set of network nodes is implemented using program state migration techniques. Service discovery and device localization mechanisms are used for defining spatial views which are collections of heterogeneous devices and over which programs iterate. SpatialViews extends Java and its programs are compiled into Java bytecode. This approach makes the language more suitable for the use on networks of more capable devices such as, for example, handhelds or laptops since most of the program heap (tens of kilobytes) needs to be communicated each time the program migrates. The authors investigate different state migration strategies for implementation of the distributed iteration operator including sequential migration along a network path, migration and state cloning using flooding waves or migration along a spanning tree topology. The language can naturally expresses actions of individual actuators but due to the sequential execution model, timely coordination of multiple actuators is not trivial.

A spatio-temporal approach to macro-programming WSN applications is explored by (Wada et al. 2008). The Space/Time-Oriented Programming language (STOP) extends the Ruby programming language to provide a model in which spatial regions of network nodes can be defined for use over a given time span. An extensible set of predefined data collection operations is used to query sensor data over specified periods of time and a temporal database is used to log the collected sensor data for future reference. Macro-program execution is coordinated by a central server which either queries the database when past sensor readings are needed or it dispatches a set of mobile agents to query the sensor nodes for the required information. The language lacks support in-network actuation and hence its main goal are data collection and event detection tasks we conclude that it is not well suited for WSAN applications.

Automatic application partitioning and distribution is the core of the Magnet Operating System (Liu et al. 2005) which provides applications with a single-system view of a distributed Java virtual machine. MagnetOS applications are automatically partitioned at the class-granularity and distributed across a heterogeneous ad-hoc network. Energy-consumption and system longevity metrics are used to optimise the placement and migration policies for thus created application components. Although the work targets wireless sensor network applications, the existing implementation supports only more capable devices such as, handhelds or laptops and it is not clear whether this approach translates to more resource-constrained WSN platforms. Also, the programming model offered by MagnetOS does not match well application scenarios in which multiple network nodes with duplicated functionality collaboratively perform a common task which is the case of many WSN/WSAN applications.

2.1.1.3 Other Approaches

In this section we discuss those macro-programming languages which elude easy categorisation. Woo et al. (2006) propose the spreadsheet abstraction to be used to program and manage wireless sensor networks. The authors extend Microsoft Excel, which is a popular office tool, with capabilities to embed, visualise and process sensor data streams. A tiered system architecture is proposed to be used in which WSN specific details are hidden behind a micro-server which transforms low-level data streams into a platform-independent XML-based data format. Aiming at enabling the use of WSNs to non-technical users, the authors argue that management tasks such as network discovery or (re-) configuration, as well as, basic data stream transformations can be easily expressed in the spreadsheet paradigm.

Macro-programming with Bayesian Networks is investigated by Mamei & Nagpal (2007). The authors show how Bayesian inference could be employed to realise such important WSN tasks as sensor calibration or signal classification. A number of experiments were performed in which small networks of two to four PDAs were used to calibrate and classify acoustic signals according to a global Bayesian signal model. Unfortunately, since the presented work is in its early stages any wider applicability of these ideas cannot be assessed.

Bischoff & Kortuem (2007) propose to use the macro-programming paradigm for configuration and maintenance of smart home infrastructures. The authors developed the Rulecaster language which provides a rule-based formalism allowing the programmer to partition the smart home into a number of logical spaces corresponding to, for example, different rooms and to define relationships between combinations of sensor readings and actions taken by actuators. For example, a user might be interested in an alarm being rised each time the kitchen oven is on while there is nobody at home. The Rulecaster compiler, apart from the rule specifications, requires a description of the whole smart home infrastructure to be provided so that it can partition the global program accordingly.

A market-based approach to macro-programming WSN applications is proposed by Mainland et al. (2004). In this model sensor nodes are represented as *agents* that trade services in return for virtual currency corresponding to their energy budgets. Services represent actions the agents take in order to realise the global computational goal as mandated by the user and they may include, for example, sending and reception of messages or sampling a sensor. The model does not require the agents to share any information but the current price vector which is updated infrequently what results in an emergent global network behaviour. The main goal of the work is to investigate novel approaches to development of energy-efficient applications in which the programmers are freed from the need to design energy-conservation strategies.

From among the above presented macro-programming frameworks only Rulecaster targets applications that require actuation. The other work is more experimental and focuses on some specific aspects of the operation of WSN applications. Rulecaster is the only language with explicit support for actuation. The language, however, since it targets smart home automation applications is designed for scenarios in which actuation is applied immediately, once a given configuration of events occurs. Such an event-driven approach makes it difficult to express applications which require more sophisticated sensor data transformations or feedback-driven operation.

2.1.1.4 Conclusions

This section reviewed the existing macro-programming languages in terms of their suitability for WSAN application development. We discussed their support for application timeliness and energy-efficiency, network heterogeneity and the ability to express feedback. We showed that those approaches that were designed with WSN applications in mind either focus entirely on sensor data processing or provide only a limited support for networked actuation. On the other hand, the actuation-enabled macro-programming languages do not explicitly address the need for time-synchronized actuation. Finally, none of the existing languages appear to support duty cycling what is a necessity when energy-efficient network operation is of importance.

2.1.2 Other Network-Level Approaches

Database abstractions were some of the earliest-proposed network-level programming models for the WSN domain. For example, SINA (Srisathapornphat et al. 2000) is a middleware platform proposed for the development of data-driven WSN applications. The middleware targets hierarchical network architectures and it enables applications to query sensor nodes for data using an SQL-like language, as well as, to perform more general computational tasks using a scripting language called SQTL - Sensor Querying and Tasking Language. Similarly, DSWare (Li et al. 2004) is a data-service middleware which provides the application layer with a more comprehensive set of services including real-time data delivery, reliable in-network storage and handling of unreliable or correlated sensor reports.

A similar, but a more general approach was taken by frameworks such as TinyDB (Madden et al. 2005) or Coguar (Yao & Gehrke 2002) which proposed to use a declarative language such as, for example, SQL to express queries on time-varying sensor data. In this model, queries are automatically translated into instructions for individual sensor nodes and are disseminated across the entire network. The query translator can make use of all available information on the network structure in order to optimise the query with a particular emphasis on the minimization of energy network consumption. Typically, these frameworks use a global spanning tree to connect all network nodes to a single data collection point called the *sink*. As data values are sent towards the sink, they are transformed, compressed or aggregated in order to reduce the communication overhead. This approach is particularly well suited to applications in which the network is treated as a single data source and in which processing and decision making is performed on a central server. Real-time considerations, feedback and actuation are outside of scope of this approach and are rather traded off for energy-efficient operation. For example, the TinyDB query processing middleware uses duty cycling in order to extend the network's lifetime[2]. Heterogeneous sensor hardware, however, can easily be used within the framework, since the queries are purely data centric, i.e., they are not tied to any platform specific semantics.

Heinzelman et al. (2004) propose to include QoS requirements at the early stage of of application definition. The Milan middleware is a data service which allows applications to request desired data streams through specification of a set of data variables. The variables are defined in terms of groups of sensors contributing to their values and QoS requirements (e.g., the level of certainty) need to be specified both for sensors and for their compositions (the variables). The middleware then creates an execution plan which selects and tunes application's networking protocols so that the desired QoS requirements can be met while minimizing energy consumption in the network.

The use of tiered network architectures is advocated by the authors of the Tenet system. Gnawali et al. (2006) argue that this approach significantly improves robustness and manageability of large sensor networks. In Tenet, the resource-constrained sensor nodes can execute several different applications simultaneously, each of which is represented as a linear dataflow graph of simple and reusable *tasklets*. Multi-node data fusion is performed by the master tier comprising resource-rich and more capable network nodes which, contrary to sensor nodes, can freely communicate which each other, possibly in a multi-hop fashion.

2.1.3 Group-Level Programming Models

Programming abstractions involving groups of network nodes such as, for example, k-hop neighbourhoods, network regions, spanning trees or target tracking groups address the need for collaborative in-network data processing in WSN/WSAN applications. Among the variety of proposed approaches, network neighbourhood data sharing appears to be a fundamental principle driving the design of different programming paradigms. Group-level programming models, even though they operate on concepts spanning multiple network nodes, differ from the network-level programming models in that they require explicit specification of actions performed by individual nodes. The group abstractions, therefore, serve only as a way to facilitate those application tasks which require collaboration of a group of network nodes.

Hood (Whitehouse et al. 2004) is a programming model which uses one-hop network neighbourhood data sharing as the main application communication primitive. Applications can define a range of virtual neighbourhoods which correspond to different filters on the values of shared variables called attributes. Hood was written in the NesC programming language (see Section 2.1.4) and it exposes a range of asynchronous interfaces for the creation and performing operations on the attributes, filters

[2]The Sosna programming language implements duty cycling during multi-hop stream aggregation in a similar way.

and neighbourhoods. Mottola & Picco (2006) introduce a more general definition of the network neighbourhood. The authors propose that the neighbour relation be defined on a purely logical basis through the specification of predicates on such static or dynamic properties as, for example, node's hardware capabilities or the current sensor readings. In this view, network neighbourhoods could span potentially across the whole network and the authors propose a new routing protocol in order to improve the efficiency of their approach. Similarly, Welsh & Mainland (2004) investigate programming abstractions that span beyond the one-hop neighbourhood. Abstract Regions allows programmers to define groups in terms of topological relations such as k-hop neighbourhoods or spanning trees, or in terms of such geographic properties like k nearest nodes or planar mesh topology neighbours. Shared variables are introduced as means of information exchange among nodes comprising regions and region reduction operations are used for folding all region values into a single one.

A more dynamic view is taken by Brown et al. (2007) who propose the Virtual Node programming abstraction in which groups of unreliable network nodes collaboratively implement a single virtual device. Virtual nodes are not bound to any particular network region and, contrary to the underlying networking substrate, they offer a deterministic and fault-tolerant programming model in which applications are specified as state-based, distributed programs communicating through explicit messaging. A similar idea of mobile group-based networked computation is exploited by Luo et al. (2006) who developed a programming model for applications related to tracking a number of targets moving through the network. Envirosuite is built on top of a target tracking middleware and it allows to associate arbitrary computation and actuation with the objects being tracked. The objects are labelled with abstract identifiers which form a dynamic network address space and a novel routing protocol is used in order to enable communication among different tracking groups. Envirosuite is part of a more general, real-time surveillance system VigilNet (He et al. 2006) which, among other services, permits network-wide duty cycling for energy conservation. Liu, Chu, Liu, Reich & Zhao (2003) generalise the tracking group abstraction to accommodate a wider class of collaborative signal processing applications of which target tracking is just one example. The authors propose a state-centric model in which leader-based collaboration groups are defined by specifying their scope and structure. Similarly to Envirosuite, computation is performed by group leaders and both the group leader and group members can change roles as the time progresses.

A rule-based approach to programming WSN applications is investigated by Terfloth et al. (2006). Their programming language FACTS offers an event-driven, stateful programming model in which applications consist of a set of rules defined over facts which are record-like data structures that can be transmitted among neighbouring network nodes. Although the language uses an explicit send primitive for triggering neighbourhood-based communication, reception of facts from neighbouring nodes is handled transparently what results in a communication model resembling neighbourhood data sharing. An extension of the rule-based programming model to the WSAN domain is proposed by Sen & Cardell-Oliver (2006). The authors propose to use abstract communication channels for neighbourhood communication, as well as, they introduce a periodic rule triggering mechanism to capture the timely nature of WSAN applications. The authors note also that if a TDMA-style communications were used duty-cycled operation could be scheduled by the compiler. A similar approach is taken by the DESAL programming language (Dalton et al. 2008) where programs are specified as sets of time-triggered and periodic rules. DESAL additionally introduces n-hop neighbourhood application state sharing for seamless integration of network communication with rule-based reasoning, as well as, it globally schedules rule invocations in order to facilitate synchronized actuation.

Rule-based programming of cooperative control systems is investigated by Klavins & Murray (2004) in their work on the Computation and Control Language (CCL). The authors propose to specify high-level application logic by means of guarded commands which are program state update expressions conditioned on a set of boolean expressions also defined on program state. CCL relies on implicit global state sharing and the authors argue that this approach, although abstract, enables formal reasoning about temporal program properties.

Inspired by work on deductive database systems, Chu et al. (2007) developed a programming model in which a broad-range of sensor network applications can be concisely specified using logical predicates and rules on dynamically changing set of facts. The language uses local data sharing in order to enable reasoning on facts held by neighbouring nodes and its compiler is able to generate compact code that easily fits on the existing resource-constrained mote platforms.

Rule-based configuration of WSN applications is proposed by Frank & Römer (2005). The authors observe that many WSN algorithms implement coordination strategies in which network nodes take different roles, for example, clustering algorithms partition network nodes into cluster heads and cluster members, while duty-cycling algorithms split the network into active and sleeping parts. Generic Role Assignment is a declarative formalism which allows to specify different network roles based on logical predicates defined over temporal properties such as, for example, the values of current sensor readings, node battery state or the roles taken by neighbouring nodes. The formalism was incorporated to the TinyCubus middleware (Marrón et al. 2005) which is a highly-customisable component framework for WSN applications.

Programming WSNs with mobile agents is explored by Fok et al. (2009). The Agilla middleware uses the tuple space shared memory paradigm (Gelernter 1985) in which agents communicate through reading and writing of tagged messages to a shared storage. Agilla builds upon the TinyLime middleware which provides an efficient implementation of tuple spaces tailored for multi-hop, resource constrained networks. TinyLime offers a model in which each network node has its own tuple space which is shared among agents residing on that node. Whenever two network nodes are within communication range, TinyLime provides a uniform view of both tuple spaces implementing this way neighbourhood-shared memory. Agilla agents can freely migrate and fork in the network and in order to decrease the program transmission cost a high-level virtual machine is used to specify and execute agent code. Kwon et al. (2006) take a similar approach but instead of virtual machines they propose to use a restricted subset of the Scheme language along with an interpreter embedded on every network node, as well as, they use explicit communication among agents residing on neighbouring network nodes.

2.1.4 Node-Level Programming Models

This section presents an overview of lower-level programming models which do not facilitate networked computation but rather they provide means for implementing the desired functionality in terms of actions performed by individual network nodes.

Currently the most popular and the most established WSN/WSAN software development platform is TinyOS (Levis et al. 2004), an event-driven, lightweight operating system written in the NesC programming language (Gay et al. 2003). NesC is an extension of the C programming language and it offers a model in which applications are partitioned into modules and wired together using bidirectional interfaces. Communication between the modules is asynchronous what means that time-consuming interface method calls are split into an invocation part and an acknowledgement part. This model of computation is sometimes referred to as the *split-phase* execution and it has been criticised for forcing the programmers to manually manage program concurrency what makes application development difficult and error-prone (Romer & Mattern 2004).

TinyOS defines a static component model in which components are singletons, the structure of the whole application is known at compile time and applications are compiled into monolithic binaries. SNACK (Greenstein et al. 2004) is a programming model which improves on some of these shortcomings. The language builds on TinyOS in that low-level components are implemented in NesC but, in addition, it views applications as compositions of parametrised services which can have multiple instances. These ideas eventually made their way to the second TinyOS release which now supports generic and parametrisable components (Levis, Gay, Handziski, Hauer, Greenstein, Turon, Hui, Klues, Sharp, Szewczyk, Polastre, Buonadonna, Nachman, Tolle, Culler & Wolisz 2005).

In recognition of the fact that deployment of monolithic applications may be disadvantageous in situations when sensor nodes cannot be easily accessed for reprogramming, the Impala middleware (Liu & Martonosi 2003) offers a programming model in which application components can be replaced at runtime. Impala was developed for the ZebraNet wild animal tracking project and its applications follow an asynchronous execution model which requires implementation of an adaptation interface for runtime updates.

The drawbacks of the split-phase execution model are recognised by Cheong (2007) who proposes to use the GALS (globally asynchronous, locally synchronous) programming paradigm for WSN application development. The notion of synchrony used in this work differs from the one used in the synchronous programming domain and it refers to the use of the blocking call mechanism, as opposed to, asynchronous, split-phase invocations. We will use the term synchronous flow of control in the rest of the dissertation to refer to this mode communication. The TinyGALS architecture, therefore, defines applications as compositions of high-level modules which communicate through buffered and asynchronous ports while control flow within the modules is synchronous.

SOL (Karpiński & Cahill 2007) is an imperative programming language which suppresses the split-phase execution model by offering a programming model in which synchronous flow of control is accompanied by a lightweight, static and deterministic concurrency model. Manual application state management, often found in TinyOS applications, is supported through the introduction of hierarchical state machines. SOL programs are compiled to efficient NesC code and generic mechanisms are provided for reuse of existing TinyOS libraries. A similar approach to WSN application development is investigated by Kasten & Römer (2005) in their work on the Object State Model (OSM). The authors propose a programming model in which applications are specified as hierarchical and parallel compositions of state machines but, contrary to SOL, OSM assumes event-triggered state transitions, while in SOL state changes have to be explicitly requested by the programmer.

Object oriented programming of WSN applications in Java is proposed by Simon et al. (2006). The authors developed a special-purpose implementation of the Java virtual machine which can be run directly on hardware without the need for operating system support. The Squawk virtual machine runs on memory-limited devices such as Sun^{TM} SPOTS and it represents applications as objects which are authenticated and can migrate through the network. Threading support is provided along with non-preemptive garbage collection. Levis, Gay & Culler (2005) propose the use of application-specific virtual machines as a way to lower the application's byte code size so that wireless network reprogramming can be more efficient. AVSM allows the programmers to tailor the virtual machine's instruction set to the set of high-level operations that define the application's logic. As an example, the authors show how the Abstract Regions framework (see Section 2.1.3) can be used as a basis of a virtual machine which operates on high-level spatial constructs. Additionally, in order to simplify the application development Levis (2005) developed TinyScript which is a high-level scripting language whose programs are compiled into the AVSM byte code. A similar approach is taken by Koshy & Pandey (2005) but, contrary to AVSM, VM* uses Java as the top-level language, as well as, performing an extensive byte code analysis in order to automatically generate the application-specific virtual machine definition. Also, VM* allows both the virtual machine, as well as, the application to be dynamically updated at runtime.

Programming WSN applications in plain C is considered by the authors of SOS (Han et al. 2005), MantisOS (Bhatti et al. 2005), Nano-RK (Eswaran et al. 2005) and Contiki (Dunkels et al. 2004) operating systems. The main contributions of this work are in the area of efficient implementation of various operating system techniques, hence are beyond the scope of this dissertation. However, we note that the authors of the first three of the above listed operating systems argue for the use of threading and synchronous flow of control as a way to simplify application development, while the authors of Contiki, although it supports both event-driven and synchronous flow of control, advocate for the use of a lightweight, cooperative threading library called Protothreads (Dunkels et al. 2006).

2.2 Synchronous Programming Languages

This section presents some of the most established synchronous programming languages, as well as, discussing their applications to the WSN/WSAN domain. We conclude the section with a discussion of the differences between distributed implementations of synchronous languages and the synchronous macro-programming paradigm presented in this dissertation.

Esterel (Berry & Cosserat 1985) is an imperative synchronous programming language designed with control-dominated reactive systems in mind whose examples include real-time process control systems found in manufacturing or transportation domains, supervision of complex systems, peripheral drivers or human-machine interfaces. Esterel programs consist of a collection of nested, concurrently running threads which use control constructs similar to those of imperative languages like C and whose execution is synchronized to a single, global clock. Threads communicate through instantaneous and global broadcasting of signals - Esterel's main data type. Esterel programs are statically scheduled to sequential code and are, therefore, guaranteed deterministic behaviour. This approach greatly simplifies formal reasoning about program behaviour since it abstracts away all details related to how concurrency is implemented on the target execution platform.

Lustre (Caspi et al. 1987) is a declarative language based on the synchronous dataflow model (i.e., a synchronous interpretation of block diagrams). Lustre programs are essentially sets of equations defined on stream variables. Streams can be thought of as infinite sequences of values or as functions of time. For example, by writing "$x = y + z$" we mean that at each *instant k* (or in each *time step k*) $x_k = y_k + z_k$ holds. The concept of feedback is commonly used to define streams of time-varying values or a form of persistent state and it can be expressed in the language through the use of recursive stream definitions. For example, the following program:

```
n = 0 -> pre n + 1
```

constructively defines a stream of natural numbers by adding at each instant $k > 1$ the value 1 to the stream value from the *previous* (i.e. $k - 1^{th}$) instant. The value 0 is used to initialise the process. Systems of equations and block diagrams are common modelling tools among control engineers and this led to Lustre's wide adoption as a standard tool for building safety-critical embedded systems: the SCADE suite (Esterel Technologies 2009), which evolved from the Lustre v3 compiler, was used, for example, to develop most of the safety-critical software running on board the Airbus A380 airliner.

Signal (LeGuernic et al. 1991) is also a declarative synchronous dataflow language but, in contrast to Lustre, it is designed for *open* systems, i.e., systems composed of independent but interacting components (Benveniste et al. 2003). The language's semantics are based on a mathematical model of multiple-clocked flows of data values and events. As opposed to Lustre which takes a functional point of view, Signal programs connect the flows by means of relations. The language's formal properties led to the embedding of its complete semantics in the Coq theorem prover (Nowak et al. 1998) so that formal reasoning about program properties could be performed. For example, Kerboeuf et al. (2000) verified, using this approach, a digital controller of a steam-boiler.

Esterel, Lustre and Signal are the three best known and established languages in the synchronous programming domain. Each of them offers determinism and verifiability much needed when building real-time systems. These benefits, however, come at a price of reduced expressiveness what makes them unsuitable for less demanding but more general applications. A number of extensions, therefore, have been proposed in order to explore the possibilities of applying the synchronous programming paradigm to other, more general programming models. For example, the Lucid Synchrone programming language (Pouzet 2006) extends Lustre to the functional programming domain by addition of a stream clock type system, higher-order streams and functions, parametrised state machines and events. The ReactiveML language (Mandel & Pouzet 2005) introduces dynamic process creation to the imperative programming model offered by Esterel, while the SugarCubes language (Boussinot & Susini 1998) embeds the synchronous model in Java.

In the context of wireless sensor networks, synchronous languages are suggested to be used for environment modelling and virtual prototyping. Samper et al. (2006) argue that the broadcast nature of WSN communications, as well as, the parallelism inherent in wireless networks make synchronous languages suitable for modelling the operation of individual network nodes. The authors note that, since WSN applications are sensor data-driven, the real network energy consumption may strongly depend on the environmental event patterns occurring at run time. As a result, the synchronous paradigm is proposed to be used to model both the environment and the application with the benefit of increasing the estimation accuracy of network energy consumption. These ideas are further extend by Maraninchi et al. (2008) who propose to use the Lustre language to build detailed application energy consumption models. We see this work complementary to ours and, in fact, providing supportive arguments for the use of the synchronous paradigm for WSAN application development.

Synchronous programs, as we noted in Section 1.4, can be compiled for execution on distributed, multi-processor architectures. For example, the SCADE suite includes a compiler for the Lustre programming language which can generate code for program execution on the Time-Triggered Architecture while Caspi et al. (1999) developed a more general, language-independent framework for distribution of synchronous programs. The synchronous macro-programming model, on the other hand, implements synchronous programming on a wireless network of sensors and actuators and could be considered a distributed implementation of a synchronous program. Despite the overall similarity, however, there are some fundamental differences between both approaches which are explained below.

Synchronous programs are distributed by partitioning the code into a number of subprograms for execution on different processors. The subprograms are then extended with glue code which implements inter-processor communication according to the target deployment architecture. For example, Lustre allows to annotate different streams and stream operators (nodes in Lustre's terminology) with processor identifiers and the compiler partitions the central program into a number of tasks which are statically scheduled, taking into consideration various timing constraints, for execution on the TTA architecture. Also, Delaval et al. (2008) propose a type system for automatic inference of such annotations for the Lucid Synchrone language. To summarise, the key characteristic of this approach is such that synchronous programs are mapped onto a distributed architecture which is known and well defined at compile time.

Our model takes a different approach. We introduce the neighbourhood data aggregation primitive for use as a basis of all network communications. The operator's semantics do not assume any particular network topology, they allow network nodes to decide at runtime whether they want to participate in the aggregation process and they admit communication failures. In this model, high-level communication operators can be defined and have to be explicitly used by the programmer to implement distributed computation. As a result, the same program can be run on a wide range of different network topologies, including a degenerated topology of only a single node. Additionally, the SOSNA programming language allows, as a way to facilitate programming of heterogeneous applications, to annotate certain program streams for evaluation on different device *classes*. Similarly to Delaval et al. (2008), type inference is used to automatically discover device class information for the rest of program streams so that the global program can be partitioned. Although, this resembles the typical distribution of synchronous programs, we note that our model is more general because the partitioning is performed for execution on groups of devices rather than on individual processors. Also, the support for heterogeneous execution is, in fact, orthogonal to the distributed semantics of the synchronous macro-programming model.

2.3 Conclusions

This chapter discussed the existing programming languages and frameworks for programming WSN and WSAN applications. A particular emphasis was given to the macro-programming languages because they are the most relevant to our work. We showed that none of the actuation-enabled macro-programming languages provided support duty cycling and that application timeliness was not explicitly supported. Moreover, when more general programming models are concerned, the support for duty cycling is provided only by two frameworks, i.e., TinyDB and VigilNet, which do not support more general WSAN applications and rather focus on more specific application domains.

Chapter 3

Synchronous Macro-Programming

This chapter describes the synchronous macro-programming model. The model defines all network communications in terms of finite sequences of neighbourhood data aggregation rounds whose execution is bounded in time. Then, we present the SOSNA programming language that is a stream-based, functional formalism designed for macro-programming heterogeneous WSAN applications. We show that the synchronous macro-programming model is flexible so that it can accommodate such high-level macro-programming concepts as multi-hop stream aggregation and propagation, leader-based tree topology construction or region-based computations. We also show, that network heterogeneity can be handled in the synchronous macro-programming setting and describe the SOSNA's heterogeneity model, which is based on an abstract device class hierarchy and a global static cluster topology underpinning all SOSNA applications. SOSNA uses type inference in order to minimise the burden of data type and device class annotations. We overview the language's type system and explain the global program partitioning technique, which is the basis of the language's support for heterogeneous networks. Also, important language design choices such as handling of recursion are explained in order to show that SOSNA applications conform to the synchronous macro-programming model. Finally, we discuss the applicability of the language to WSAN application development and give a range of examples in order to show the practicality of the approach, as well as, to motivate the design of the model.

3.1 The Synchronous Macro-Programming Model

Synchronous execution is the core characteristic of the synchronous macro-programming model. In fact, the model can be thought of as an extension of the synchronous programming paradigm in which

```
foreach instant do
   read_inputs();
   compute_outputs();
   write_outputs();
end
```

Figure 3.1: Abstract implementation of a synchronous program.

network neighbourhood communication is allowed to take place within a single time *instant*. We proceed, therefore, with an informal description of the synchronous programming paradigm and then follow with the presentation of the synchronous macro-programming model.

The synchronous programming paradigm is build around the central notion of the global logical clock that defines program execution as an infinite-in-time sequence of *instantaneous* reactions to an infinite sequence of input values. In this view, inputs correspond, for example, to sensor readings sampled at regular time intervals and reactions may represent actions taken by actuators at the rate defined by the global clock. The synchronous hypothesis states that all computations taking place between reading of an input and executing a reaction take a negligible time, hence are considered instantaneous. Such a formulation, although natural in many control frameworks, is not a particularly practical one. The hypothesis, therefore, is used as an idealised system model to simplify reasoning about program behaviour. In practise, instant duration is entirely application-specific and it is mainly required that all computations finish before the instant ends (Potop-Butucaru et al. 2005). As a result, synchronous programs can be viewed as iteratively applying in every instant a reaction function whose task is to transform the inputs into the corresponding outputs. For example, Figure 3.1 presents an abstract and generic implementation of a synchronous program in which reactions are periodically calculated by the `compute_outputs` function. Also, since many synchronous programming languages define their semantics recursively, the same logic applies to program components (i.e., units of concurrency) and it is up to the compiler to find the appropriate scheduling of computations and communications so that the abstract semantics can be realised.

Synchronous programs communicate through instantaneous broadcasts of signals that, depending on the language, may carry data values or may be abstract signalling concepts. In the context of the synchronous hypothesis, instantaneous broadcasting means that all program components awaiting a signal can read its value at the same time, every time it is broadcast. But since real-world implementations assume that instants have non-zero duration, instantaneous broadcasting is understood as the ability of all program components to read the signal value in the same instant in which it is emitted. Additionally, it is required that the reading be consistent, i.e., either all components see the signal as *present* or they all see it as *absent* throughout the instant duration, hence, making it is possible to decide upon signal absence.

The difference between the abstract notion of an instant and its implementation as a time interval of non-zero duration is crucial to the understanding of the synchronous macro-programming model. We use, therefore, the following definition to clarify the concept.

Instant is a unit of the abstract, logical and discrete time that drives the operation of synchronous programs. Within one instant, all program components read their inputs, communicate and write their outputs, advancing this way the overall program state by one logical step.

To illustrate these ideas we will use the Lucid Synchrone language (Pouzet 2006) as an example since it was the main inspiration for SOSNA. In Lucid, programs are composed of streams that are infinite-in-time sequences of data values that can be either present or absent in which case they are denoted with a special symbol ·. The abstract signals correspond in the language to stream data values and broadcasting corresponds to passing values between streams that reference one another in very much the same way as variables reference other variables in traditional programming languages. The following example program defines the stream `y` in terms of values taken by the stream `x`. Numerical literals represent in this program constant streams and the `if` and `>` operators are evaluated by iteratively applying them to consecutive stream data values throughout the time.

```
y = if x > 3 then 1 else 0
```

The following table shows how both streams are evaluated (we assume that `x` is defined elsewhere or that it corresponds to a program input). Note that there is no delay between reading of the current `x` value and the computation of the current value of `y` so, despite the logical dependency, both streams are considered to be evaluated at the same time, hence are synchronous.

Stream	**Values**						
`x`	1	2	3	4	5	6	...
`y`	0	0	0	1	1	1	...
`3`	3	3	3	3	3	3	...
`1`	1	1	1	1	1	1	...
`0`	0	0	0	0	0	0	...

Absent values are not directly representable in Lucid and, instead, are handled implicitly. The language takes a conservative approach and requires that stream value presence/absence times can be established at compile time so that the compiler can check whether program streams can be safely composed. For that purpose, Lucid associates with every stream a logical clock that can be thought of as a periodic stream of Boolean values representing stream presence and absence times. Consider, for example, the situation depicted in Table 3.1 in which two streams are driven by complementary clocks. It is not possible, in the synchronous semantics, to apply point-wise arithmetical operations to those streams because in every instant the value of one of the operands would be missing. Thanks to the ability to decide on value presence or absence, however, it is possible to meaningfully merge both streams into one that does not comprise absent values.

The synchronous macro-programming model extends the synchronous programming paradigm with the use of network neighbourhood communication as a primitive and atomic (non-divisible) operation. In contrast to the abstract communication among components in synchronous programs, network neighbourhood communication refers to message exchange among neighbouring network nodes over (wireless) communication interfaces. In the rest of the thesis, we will use the term communication in accordance with its usual meaning, i.e., message exchange over a communication interface, and not as it is often understood in the literature on synchronous programming.

General neighbourhood communication defined as network-local, connection-less and broadcast-based message exchange is a basic building block of many WSN/WSAN applications. For example, programming frameworks and languages such as Hood, Agilla or Desal use periodic neighbourhood communication to implement their core data sharing services while the Token Machine Language (an intermediate program representation language developed for the Regiment macro-programming language, see Section 2.1.1.1), builds upon a neighbourhood communication operator to implement multi-hop network gradients and region-wide data aggregation. Proto and Flask are two macro-programming languages that use a neighbourhood data aggregation operator as a basic building block of complex network communication processes that may potentially span across network regions of arbitrary size. Beal & Sussman (2005) show additionally that, assuming periodic execution, Proto's neighbourhood data aggregation operator can be used to concisely express a wide range of biologically-inspired amorphous programs in a purely functional way.

The synchronous macro-programming model uses neighbourhood data aggregation as a basic build-

Stream	**Values**					
`x`	x_0	$\cdot$	x_2	$\cdot$	x_4	...
`y`	$\cdot$	y_1	$\cdot$	y_3	$\cdot$	...
`merge c x y`	x_0	y_1	x_2	y_3	x_4	...
`c`	true	false	true	false	true	...
`not c`	false	true	false	true	false	...

Table 3.1: Merging of complementary streams in Lucid.

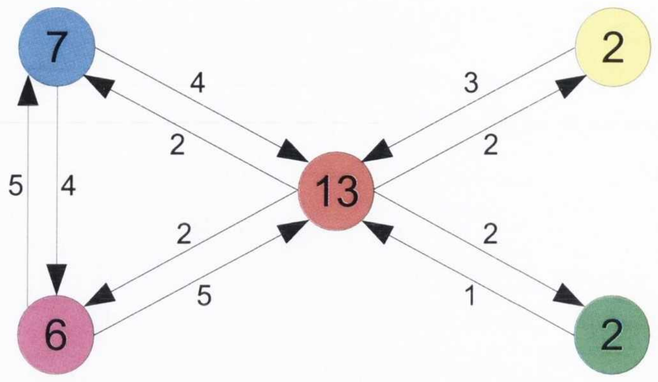


Figure 3.2: One round of neighbourhood data aggregation.

ing block of higher-level and more complex network communication schemes. This means that within one instant a number of neighbourhood data aggregation operations might need to be performed. Following the abstract formulation of the synchronous hypothesis, we define neighbourhood data aggregation as follows.

Neighbourhood data aggregation is a simultaneous, broadcast-based exchange of data items among neighbouring network nodes in which each of the participating nodes contributes at most one data item and which results in computation of an aggregate value by at least some of the participating nodes.

In order to make the model as general as possible, we assume the following about the implementations of neighbourhood data aggregation:

1. data items may be lost or delivered selectively,
2. the order of data item delivery can be arbitrary,
3. data items may not be duplicated,
4. data items delivered after the round has ended are discarded.

Figure 3.2 depicts an example neighbourhood data aggregation operation in which each of the nodes calculates the sum of the received data items while sharing its own data item with its neighbours. Communication links in the figure follow a local network topology defined, for example, by physical wireless connectivity while multiple arrows stemming from the nodes represent broadcast transmission.

Similarly to how the synchronous hypothesis is formulated, we have to address the imperfections of real system implementations. If timeliness of program execution is to be achieved, issues such as wireless medium unreliability, delays resulting from wireless channel sharing, possible node failures or network topology changes lead to the requirement that each neighbourhood data aggregation operation must be realised within a time period of bounded duration. Therefore, we introduce the notion of a round of neighbourhood data aggregation.

Round of neighbourhood data aggregation (or simply a round) is a time period of known and bounded duration that refers to the duration of one neighbourhood data aggregation operation in real time.

The exact value of round duration is application-specific because different network configurations may have different timing properties, as well as, different applications may have different timing requirements.

Bounding the duration of data aggregation rounds implicitly brings the notion of common network time because the participating nodes must know when they are required to engage in communications. The synchronous macro-programming model, therefore, requires the provision of a global network time reference that is used to define the logical application clock and this requirement, as we showed in Section 1.2, does not pose a significant challenge to the existing WSAN technologies.

Because neighbourhood data aggregation is considered a primitive and atomic operation, multiple aggregation operations can take place within one instant. In terms of the abstract synchronous program implementation from Figure 3.1, this corresponds to the ability to use neighbourhood data aggregation within the `compute_outputs` function. As a result, the real time needed to execute one instant of program operation is non-negliglible. Therefore, in order to bound the amount of time needed to execute one program instant, we restrict the use of data aggregation, rejecting those synchronous macro-programs for which the total number of data aggregation operations taking place within one instant cannot be established at compile time. As a result, implementations of synchronous macro-programs are periodic compositions of a fixed number of data aggregation rounds.

Communication Cycle is a scheduling of all program neighbourhood data aggregation rounds that directly corresponds to the execution of one program instant.

The model does not enforce any particular communication scheduling strategy and it is only required that the standard data dependency constraints are satisfied, i.e., the results of one data aggregation round can be used after the round has finished. Figure 3.3 presents an example communication cycle: note that it is possible to schedule several data aggregation rounds for execution in parallel and that it is possible to introduce additional delays that may be used, for example, for computation or duty cycling.

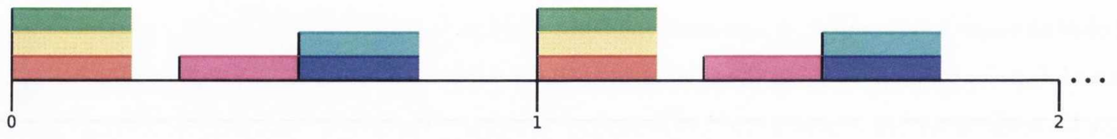


Figure 3.3: Two communication cycles comprising six data aggregation rounds each.

The scheduling of application's communication cycle as a static composition of bounded-in-time data aggregation rounds must be established at compile time and, therefore, must be obeyed by all network nodes at runtime. Such a global approach to network communication scheduling has an important consequence: within one instant, network nodes can propagate information by only a bounded number of hops and this distance can be established at compile time since multi-hop communication can only be implemented in the model as a sequence of neighbourhood data aggregation rounds. The synchronous macro-programming model, therefore, takes a static approach, in which some network parameters such as, for example, the maximum extent of the underlying network topology must be known at compile time. For example, the SOSNA programming language introduces a compilation parameter that denotes the maximum depth of network spanning trees used to implement multi-hop data aggregation. This way, program text may be freed of assumptions on the deployment network topology without breaking the model's central requirement.

Although this might be limiting to some applications, the reliability and timeliness considerations suggest, as we discussed in Chapter 1, that smaller-extent topologies are preferable in the WSAN domain. On the other hand, global and periodic communication scheduling carries a number of important benefits: application reaction time, i.e., the duration of one instant, can be established at compile time and tuned to reflect application-specific requirements while sensor sampling and actuation can be globally synchronized. Finally and most importantly, global communication scheduling gives way to automatic duty-cycling, which is a crucial characteristic of energy-efficient WSAN applications: sleep times can be automatically inserted into the global schedule and because network nodes know the durations of data aggregation rounds, they can choose at runtime to switch their radios off if, based on the current application state, they conclude that they can neither provide nor receive information relevant to the result of a given data aggregation round. In fact, it is exactly this property that allows the SOSNA compiler to generate application code capable of energy-efficient execution.

3.2 The SOSNA Programming Language

SOSNA is a stream-based functional language, originally inspired by the synchronous programing language Lucid Synchrone (Pouzet 2006). SOSNA is also a synchronous macro-programming language, meaning that it implements the synchronous macro-programming model. The language targets potentially heterogeneous and resource-constrained WSAN applications and its compiler can generate application code that runs in bounded memory and time, as well as, which is capable of duty cycling. SOSNA applications require a runtime network time-synchronization service for correct execution and, provided that the underlying operating system can guarantee real-time bounds on processing and wireless channel access, SOSNA applications have bounded reaction times and implement synchronized sensor sampling and actuation. The following sections describe the core features of the language and Section 3.4 presents a range of more advanced application examples discussing the language's applicability to WSAN application development. The description of how SOSNA applications compiled by the current prototype compiler execute on wireless sensor networks are deferred to Chapter 4.

3.2.1 Functional, Stream-Based Programming

Streams, i.e., infinite-in-time sequences of values are the core concept of the language. SOSNA offers a synchronous stream transformation model that is similar to that of Lucid Synchrone. Programs are defined as static compositions (networks) of stream operators, which are evaluated in synchrony in essentially the same way as it was presented in Section 3.1. Contrary to Lucid, SOSNA does not allow to infer stream value presence/absence patterns at compile time because stream value presence might depend on the results of neighbourhood data aggregation and hence it possesses a degree of uncertainty related to possible communication failures. In the rest of the presentation, we will use the the term *scalar values* to represent individual stream data values and the term *null values* to represent the $\cdot$ symbol, i.e., the absence of a stream data value in a given instant. Also, we will use the term *stream values* whenever the distinction between scalar and null values is not relevant.

Constant streams are SOSNA's most basic streams. They are represented by literals of various sorts such as integers, floating point numbers or strings. Constant streams are the equivalent of constant values in traditional programming languages and they represent streams whose values never change. Analogously to auxiliary variables in functional languages, auxiliary streams can be defined using the **`let`** expression for local declaration scoping or using the global stream definition form. The following example defines two local, constant streams `pi` and `x`, and a global stream `s`. Note that SOSNA does not require providing type annotations - all types are automatically inferred by the compiler.

```
(let (pi 3.1415)
  (let (x 2.0)
    (* x pi)))
(def s "A global and constant stream")
```

SOSNA uses the prefix notation of S-expressions, which is the basis of the LISP and Scheme programming languages (see, for example Abelson & Sussman 1996). In this notation, language expressions are either literals, identifiers or lists of other expressions surrounded by a pair of parentheses. The first sub-expression of a list usually represents a language operator or a function while the other sub-expressions are its arguments. Therefore, the inner-most **`let`** operator from the above program can be read as follows: let `x` equals `2.0` in the expression `(* x pi)`, i.e., multiply `pi` by `2`.

The distinction between streams and the values that comprise them is fundamental in SOSNA and the language, as a result, can be partitioned into two distinct domains: the functional domain comprising functions, scalar variables and scalar constants, and the stream domain comprising streams, stream operators and other high-level constructs that are described in the following sections. Functions in SOSNA are pure, i.e., they are side effect-free and since they are first-order, meaning that they cannot be passed as arguments to other functions or returned as values, streams of functions cannot be defined in the language. The purpose of functions in SOSNA is to define scalar value transformations and, similarly to other functional languages, anonymous and recursive functions can be defined. The following code snippet shows example definitions of named and anonymous functions. The first function calculates the minimum of two values while the second one is anonymous and it applies the `min` function in order to compare its only argument `z` to the value 2. Note that the syntax of function application is the same as that of predefined operators and that the **`if`** operator does not require the *"then"* keyword - instead, the last two arguments represent both alternatives.

```
(def-fun (min a b)
  (if (< a b) a b)
(lambda (z)
  (min z 2))
```

Functions can be applied to streams point-wise in time using the **`smap`** operator, which is the most fundamental stream operation in SOSNA. The operator applies a given function to each value of all of its argument streams and returns a stream of results of these applications. The semantics of this process are the following: in every program instant the result of the operator's application is present if and only if the values of all argument streams are present in the same instant. The following program defines a stream that is a running minimum of two other streams and Table 3.2 presents its example

Stream	**Values**							
`x`	1	4	3	·	1	·	3	...
`y`	3	2	1	1	·	·	2	...
`(`**`smap`**` min x y)`	1	2	1	·	·	·	2	...
`(`**`smap`**` (`**`lambda`**`(z) (min z 2)) x)`	1	2	2	·	1	·	2	...
`(`**`smap`**` < x y)`	`true`	`false`	`false`	·	·	·	`false`	...

Table 3.2: Example evaluation of the `smap` operator.

evaluation. The figure also shows that standard arithmetical, logical and relational operations such as, for example, the < operator, are treated in exactly the same way as user-defined functions.

```
(def stream_min (smap min x y))
```

In order to simplify definitions of basic stream transformations, Sosna allows to omit the **smap** operator when the built-in arithmetical, logical and relational operators are used. The following two stream definitions are equivalent and the compiler implicitly adds **smap** operator instances when they are omitted. Both streams evaluate to a constant stream of the value 125.

```
(def x (* 5 (+ 10 15)))
(def y (smap * 5 (smap + 10 15)))
```

Note that the semantics of arithmetical expressions when considered outside of their enclosing program context can be ambiguous. Consider, for example, the expression `(+ 10 15)`, it can correspond to the addition of two scalar values or it can be an implicit **smap** application to two constant streams. Sosna distinguishes between such cases by taking into consideration the enclosing program context. Therefore, the language defines two expression definition contexts: the *function context* and the *stream context*. The former corresponds entirely to function bodies and the latter to all other parts of the program. We will use the simplified function mapping notation whenever its definition context can be clearly established.

Null values can be implicitly introduced to programs through the use of stream filtering. The **within** operator takes a stream of Boolean values - the guard stream, as its first argument and one stream of an arbitrary type as its second argument. The operator returns a stream that is equal to the operator's second argument whenever the guard stream takes the value `true`. In all other cases, i.e., when the guard stream takes the value `false` or when it takes the null value, the result stream takes the null value. The following program fragment demonstrates how stream thresholding can be realised. In short, the program filters out those values of the stream `(* 2 x)` that are evaluated in the case in which the stream `x` takes values greater than 5. An example evaluation of this program is presented in Table 3.3.

Stream	Values								
`x`	2	4	6	·	2	4	6	·	...
`x > 5`	`false`	`false`	`true`	·	`false`	`false`	`true`	·	...
`thresh`	·	·	12	·	·	·	12	·	...
`(merge 0 thresh)`	0	0	12	·	0	0	12	·	...

Table 3.3: Example evaluation of the stream filtering program from page 43.

```
(def thresh
  (within (> x 5)
    (* 2 x)))
```

As we said in Section 3.1, the synchronous programming paradigm requires that it is possible to make decisions based on stream value absence. To this end, SOSNA uses the **merge** operator that allows to substitute the values of one stream for the null values of another. The following program defines a stream that takes the value 0 each time `thresh` takes the null value. The results of an example evaluation are also presented in Table 3.3.

```
(def s (merge 0 thresh))
```

The Composition of **within** and **merge** operators can be used to define a form of the conditional **if** operator for use in the stream domain. Normally, the operator cannot be applied to streams in the same way as, for example, functions are because its semantics involve selective evaluation of its alternatives. Hence, the **if** operator can be applied only to scalar values within the function context. The stream programming paradigm assumes, on the other hand, that programs are networks of stream operators through which values are constantly flowing. Therefore, it is not possible to decide at runtime that some streams should not be evaluated. Instead, programmers are concerned with directing stream values through the operator network in such way that the desired computational result is achieved. We use the following program to explain the difference between the use of stream filtering operators and the use of the functional **if** operator.

```
(def a (merge y (within (< x y) x)))
(def-fun (min x y)
  (if (< x y) x y))
(def b (smap min x y))
```

As long as both `x` and `y` take non-null values, the `a` and `b` streams are computationally equivalent. The difference, however, manifests itself when the values of either `x` or `y` are absent. In the first case, `a` remains present as long as `x` and `y` are not absent simultaneously, while `b` takes non-null values

Stream	Values							
`x`	1	2	·	2	1	·	1	...
`y`	2	1	2	·	2	·	2	...
`(smap min x y)`	1	1	·	·	1	·	1	...
`(merge y (within (< x y) x))`	1	1	2	2	1	·	1	...

Table 3.4: The relation between the use of stream filtering operators and the use of the `if` operator.

only when both `x` and `y` are present. Table 3.4 presents an example evaluation of the program and Figure 3.4 depicts both stream definitions in form of stream operator networks.

The stream operators presented so far allow to express stream computations in which only the current stream values are taken into consideration. In the operator network terminology, such programs correspond to acyclic operator networks. However, in many practical applications it is beneficial to be able to refer to stream values computed in instants preceding the current one. Consider, for example, a program that detects changes in the consecutive values of a given stream. It would be impossible to implement such behaviour without access to the stream's past state. SOSNA follows the approach taken by Lustre and Lucid Synchrone and it offers a stream *delay* operator named **`fby`**. The operator simply defines a new stream whose current value is the one that the stream that is its second argument took in the previous instant. **`fby`** takes, therefore, two arguments: one providing the initial value for the resulting stream used in the first instant of program execution, and the other one being the stream to be delayed. The operator's name is an abbreviation of the phrase "followed by" and it corresponds to the operator's semantics: the initial value of the result stream is followed by all values of the delayed stream. The following program implements stream value change detection:

```
(def change? (smap != x (fby 1 x)))
```

The `!=` function implements the "not equal" operation and the stream `change?` takes the value `true` whenever the value of `x` changes and the value `false` otherwise (Similarly to the Scheme programming language, SOSNA allows such characters as `?` or `-` be part of program identifiers). Table 3.5 presents an example evaluation of this program: note that null stream values are delayed in the same way as scalar values but the program does not consider them as a change. In order to achieve such effect the

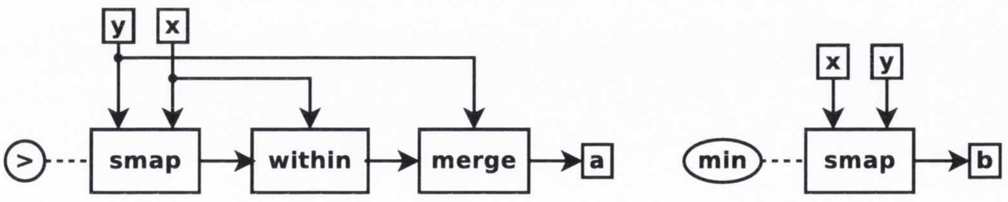


Figure 3.4: Stream operator network representation of the example stream filtering programs.

Stream	Values									
x	1	1	2	2	2	1	1	·	1	...
(**fby** 1 x)	1	1	1	2	2	2	1	1	·	...
(**smap** != x (**fby** 1 x)	f	f	t	f	f	t	f	·	·	...

Table 3.5: Example evaluation of the fby operator for change detection (abbreviations are used for the `true` and `false` values).

merge operator would have to be used.

Following Lustre and Lucid Synchrone, SOSNA allows to define recursive streams that can be used to implement persistent state. Consider an application that needs to integrate (sum) all values of a given stream. Such program would need be able to accumulate consequent stream values. Recursive stream definitions together with the **fby** operator can be used to achieve this effect. The following program defines a stream that counts from 1 to infinity, increasing in every instant its previous value by one:

```
(def-rec n (+ 1 (fby 0 n)))
```

The next definition uses an auxiliary stream `change_value` to convert the Boolean values of the previously defined stream `change?` into integer values so that they can be summed by the recursive stream `sum_of_changes`.

```
(def change_value (smap (lambda (x) (if x 1 0)) change?))
(def-rec sum_of_changes (+ change_value (fby 0 sum_of_changes)))
```

Similarly to the first definition, `sum_of_changes` is iteratively evaluated by adding the current value of `change_value` to the previous value of `sum_of_changes`. The streams take the following values:

Stream	Values									
n	1	2	3	4	5	6	7	8	9	...
change?	f	f	t	f	f	t	f	·	·	...
change_value	0	0	1	0	0	1	0	·	·	...
sum_of_changes	0	0	1	1	1	2	2	·	·	...

Note that null values propagate and unless the **merge** operator is introduced, `sum_of_changes` stays null forever, starting from the first instant in which `change?` takes the null value. An example improvement might replace all null values taken by `change_value` with a default value:

```
(def-rec sum_of_changes (+ (merge 0 change_value)
                           (fby 0 sum_of_changes)))
```

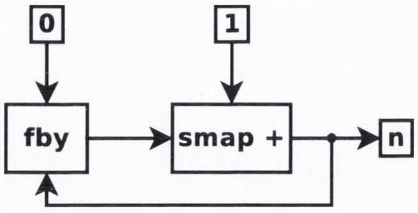


Figure 3.5: Counting to infinity through recursive stream definition.

The above examples showed that recursive streams are constructed by referencing them within their own definitions. This approach is sound because SOSNA uses the **fby** operator to delay these recursive references or, in other words, to make them refer to values already computed in past instants. Because recursive definitions such as `(def-recx x)` would be difficult to apply meaning to, SOSNA enforces all recursive stream references to be delayed.

Functions are the most fundamental way of creating reusable code in all functional languages. Since SOSNA uses functions to abstract transformations of scalar values only, a complementary concept is needed for the stream domain. To this end, the language introduces the notion of user-defined stream operators, or alternatively stream functions, which are direct analogs of scalar functions for use with all non-scalar constructs of the language. The distinction between scalar and stream functions allows to separate both domains with the main benefits of confining function recursion to the scalar domain and being able to use functions written in other programming languages. The following example program demonstrates named and anonymous stream function definitions whose syntax, with the exception of the initial keyword, is exactly the same as that of their scalar counterparts.

```
(def (IF cond x y)
  (let (a (within cond x))
    (merge y a)))
(big-lambda (init)
  (letrec (count (+ 1 (fby init count)))
    count))
```

Both stream and scalar functions can use the **let** operator to introduce auxiliary stream and variable definitions respectively. The **letrec** operator has a similar purpose and syntax but it is used for auxiliary recursive definitions. The first stream function implements the earlier defined form of the **if** operator for streams (see Figure 3.4) while the second one returns a stream that counts to infinity starting from a given value (the first value of the `init` function parameter).

We conclude this presentation of SOSNA's core concepts with an overview of tuples, which are a common way, in many functional programming languages, of implementing anonymous records or, in other words, compositions of values of different types. Formally, tuple is a product type of a fixed number of other data types, including tuples themselves. Sosna allows to use tuples in both stream and functional domains but the semantics are slightly different in each case. In the functional domain, tuples have the usual meaning, i.e., conglomerates of values. For example, the following scalar function defines two tuples, one being a composition of two integers and the other combining the first tuple and a floating point value.

```
(lambda (x)
  (let (t1 (tuple 1 2))
    (tuple t1 2.0)))
```

Tuples are ordinary values, hence they can be passed as arguments to functions and returned as results. Sosna, currently, allows to access the tuple's composite values only through pattern matching, which is a technique common in many functional languages. The following example shows how it works for both function arguments and auxiliary `let` definitions:

```
(def-fun (vector_add (tuple x1 y1) vec2)
  (let ((tuple x2 y2) vec2)
    (tuple (+ x1 x2) (+ y1 y2))))
```

The program defines a function of two arguments, both of which are two-element tuples. The first argument does not have a name but instead is matched against two variables corresponding to each of its elements - `x1` and `y1` respectively. The second argument `vec2` is also a tuple, but its elements are assigned to the `x2` and `y2` variables in the following `let` definition. The function returns, as a result, a tuple of sums of the first and second elements of the input tuples respectively, implementing this way a form of two-dimensional vector addition. Sosna's compiler efficiently translates tuples into C structures and pattern variables are implemented as C structure field references.

The same syntax for tuple creation and matching is used for the stream domain but, contrary to the functional domain, tuples can only be applied to streams (as opposed to other constructs) and are understood as streams of tuples. This, among other consequences, means that stream tuple definition is regarded as the point-wise in time application of the scalar tuple constructor to the consecutive values of the argument streams in the same way as arithmetical operations are performed on streams. The following program defines three constant streams of tuples:

```
(def v1 (tuple 1 2))
(def v2 (tuple 3 4))
(def x y (smap vector_add v1 v2))
```

3.2.2 Building WSAN Applications

All of the programs presented in the previous section described abstract computations that had no contact with the outside world. SOSNA targets WSAN applications, therefore, sensing and actuation are the only means applications may use for interaction with the outside world. The language addresses sensing and actuation through the notions of abstract program inputs and outputs, the semantics of which are defined by the underlying platform-specific libraries. The simplest, yet fully-functional SOSNA program is perhaps the mythical "Hello World" application:

```
(def-class PC)
(output (Display (at PC "Hello World")))
```

The **def-class** keyword defines an abstract device class called `PC`. Device classes serve as a way to address applications that are meant to be run in networks comprising several different classes of nodes, for example, sensor, actuator and data processing nodes, and they are described in more detail in Section 3.2.3. From an abstract point of view, the above program sends in every instant consecutive values of the constant stream `"Hello World"` to the `Display` actuator that is located on devices belonging to the abstract `PC` class. Naturally, the actuator can be implemented differently on different platforms and unless some form of standard libraries are developed, the programmers must provide an implementation of the actuator's driver written in some lower-level language. This approach ensures generality and extensibility of the language and it taken by several other macro-programming languages such as, for example, Flask, Macrolab or COSMOS (Mainland et al. 2008, Hnat et al. 2008, Awan et al. 2007).

Sensors are represented in SOSNA by abstract program inputs that can be thought of as streams of time-stamped scalar values. Inputs are allowed to take null values in order to model event-based operation of some sensors (for example, motion detectors), as well as, to enable sensor state control, which is an important characteristic of many resource constrained WSN/WSAN applications. Inputs are not treated as ordinary streams because, in addition to sensor readings, they carry time stamps that represent the real time in which the sensor reading was taken. Therefore, stream operators cannot be used directly to transform inputs and the language provides two special-purpose operators **vals** and **time** for extraction of input's data values and timestamps[1]. The following program gives an example of how the operators can be used. Note that inputs are defined by specifying the device class on which they reside and the data type of their readings, as well as that there is no need to specify the device class of the displayed stream of tuples. Device classes are part of the type system and the

[1]Equivalently, the inputs could be defined as streams of tuples of data values and their timestamps but because of aesthetical reasons, we decided to use a special-purpose concept.

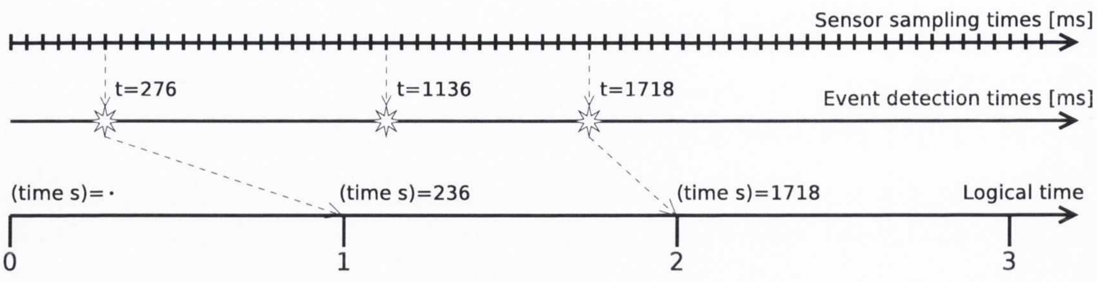


Figure 3.6: The relation between the real time of detection events and the logical application time.

compiler can infer them in most cases. Here, since the device class information is initially associated with the `light_sensor` input, there is no need to add additional annotations.

```
(def-class MicaMote)
(def-input (light_sensor @ MicaMote : int))
(output (Display (tuple (vals light_sensor)
                        (time light_sensor))))
```

The main reason behind associating temporal information with input data values is to be able to keep track of the real time in which asynchronously operating sensors detect events in the application scenarios in which the logical application clock ticks at relatively low rates. It is expected that, unlike in the case of many other embedded systems, WSAN applications will be operating at comparatively lower rates because of their need to transmit information wirelessly, as well as because of the likely need to duty-cycle their operations. Hence, the duration of program cycles in SOSNA applications may be large, compared to sensor sampling rates required by some applications such as, for example, sound processing.

In order to support applications in which either sensors need to be sampled at high rates or the real time of event detection is important, SOSNA requires input drivers (analogous to the output drivers) to provide the detection time stamp. High-frequency sensor sampling, therefore, has to be realised within the sensor driver and only high-level detection events can be passed to the application. If the driver detects more than one event within a single application cycle, only the most recent is considered as the input value for the subsequent instant. The main reason for such an approach is to guarantee bounded memory execution. Hence, unless more complex data types such as arrays are used for input data values, SOSNA applications can receive in every instant at most one reading from from each sensor. Figure 3.6 presents an example scenario describing this idea.

SOSNA programs are intended for execution in time-synchronized wireless networks of multiple devices with sensing/actuation capabilities. The language's semantics require that, programs are executed at exactly the same rate across the entire network so that logical clocks of individual network nodes can run in lock-step. We allow network nodes to start program execution at arbitrary times, as well as, to restart the program when time synchronisation is lost for some reason. As a result, although program streams are evaluated in synchrony, different nodes may experience a shift in *state*. In the rest of this section, we assume, for simplicity, that all network nodes are of the same type, i.e., they belong to the same device class, and that they are running exactly the same program.

Since it is difficult to start all network nodes at the same time, the shift in the state of stream evaluation is likely to occur in many applications. This is not, however, a significant problem because the execution of SOSNA applications is expected to be data-driven, i.e., to depend on the the values of regularly changing sensor readings and, hence, naturally correlate data processing among neighbouring nodes. We use the following program to explain the idea.

```
(def-class node)
(def-input sensor @ node : int)
(def-rec n (+ 1 (fby 0 n)))
(def detected
  (within (> (vals sensor) 100)
    true)))
(output (Blink detected))
```

The program defines two streams: `n` counts from 1 to infinity and `detected` takes the value `true` whenever `sensor` takes a value greater than 100. Network nodes running this program will blink their LEDs each time `detected` holds. Figure 3.7 presents an example networked execution scenario and the following table shows how three neighbouring nodes, each of which starting at a different time, would execute the program given a particular pattern of sensor readings.

Stream	**Values**							
node A								
n	0	1	2	3	4	5	6	...
detected	·	·	·	·	t	·	·	...
node B								
n	·	0	1	2	3	4	5	...
detected	·	·	·	·	t	·	·	...
node C								
n	·	·	·	·	0	1	2	...
detected	·	·	·	·	t	·	·	...

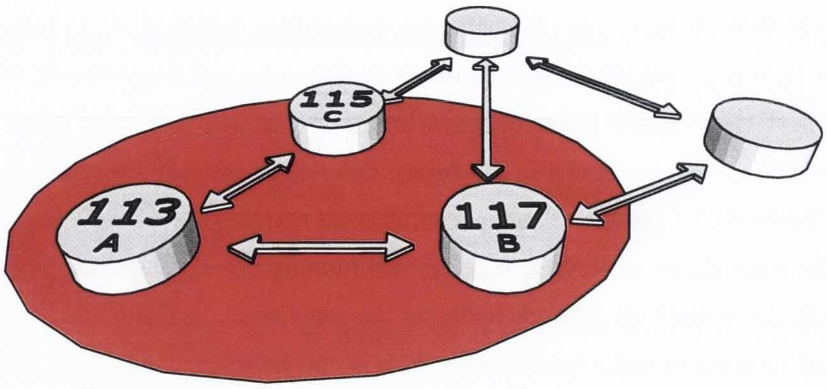


Figure 3.7: Networked streams as dynamic regions.

As we can see, the shift in state occurs for the values of `n`. This is because `n` is a recursive stream and its definition is purely computational. It is possible, however, to make recursive streams and persistent state more dynamical. For example, the following stream counts the number of instants during which the `detected` stream reports `true`. We reuse the `IF` stream function definition presented in the previous section.

```
(def-rec detect_n (+ (fby 0 detect_n)
                     (IF detected 1 0)))
```

Networked execution extends the stream semantics with an additional spatial dimension. Figure 3.7 shows that every stream can be thought of as a dynamically changing spatial region defined by the values it takes across the network. Region membership in each instant can be defined as the set of nodes at which the stream takes non-null values. Indeed, this is the main reason for which the **`within`** stream operator received its name - the operator can be used to limit one region to the set of nodes located *within* another region and, as a result, its semantics involve both temporal and spatial filtering of stream values. Spatial regions have been shown to be a natural abstraction for many WSN applications and are used by several macro-programming languages such as, for example, Regiment and Proto (Newton & Welsh 2004, Beal & Bachrach 2006). In the rest of the chapter, we will use the region metaphor interchangeably with the local view in order to simplify presentation.

The fact that region membership may change in subsequent instants is an important property of the language allowing SOSNA programs to adjust to changes in the underlying communication topology. Although this approach is common among all functional macro-programming languages, care must be taken to ensure that algorithms that rely on region-based computations compute the desired results. This, however, cannot be guaranteed by the compiler since the possible levels of network instability cannot be predicted at compile time. Therefore, the language semantics state that throughout the duration of one program instant (the cycle) network topology is considered constant. Apart from that, SOSNA takes the best-effort approach in which the system runtime is responsible for maintaining the most stable network topology and the most reliable communications possible throughout the entire program execution time. We discuss alternative implementation choices in Section 4.2.

Single-hop Stream Aggregation

Neighbourhood data aggregation is the simplest form of networked computation that can be implemented in SOSNA. The synchronous macro-programming model states that neighbourhood data aggregation is an atomic operation that must be completed within one instant, hence, the language defines it as an elementary stream transformation. We use the term *neighbourhood stream aggregation* to refer to the process that can be thought of as a point-wise application of neighbourhood data aggregation to the values of a given stream. The process involves, in essence, three core components:

- *aggregated stream* - the stream whose values are to be aggregated,
- *initialisation stream* - the stream whose values are used to initialise the result of aggregation on each of the participating nodes,
- *aggregation function* - the scalar function used to combine the values received from the neighbouring nodes with the initial aggregation value in order to compute the aggregation result.

The operational semantics of neighbourhood stream aggregation are the following. In every instant:

1. all nodes at which the aggregated stream takes a non-null value (the sending nodes) broadcast this value to their neighbours,
2. all nodes at which the initialisation stream takes a non-null value (the receiving nodes) accept the aggregated stream values from their neighbours and use the aggregation function to compute the aggregation result for this instant.

Note that the sets of transmitting and receiving nodes do not have to be equal and that the programmer has implicit control over sending and receiving through the manipulation of the aggregated and initialisation streams.

Within one round of data aggregation, the participating nodes may receive several scalar values from their network members. We require that these values come one at a time, although their relative order is not specified. Consider an arbitrary instant t and a receiving node n: let i_n^t be the value the initialisation stream takes at time t, at the node n, let $\{x_i^t\}_{i=0}^{N_t}$ be the set of values n has received from its neighbours at time t and let $f(\cdot,\cdot)$ be the aggregation function. We define the result of neighbourhood stream aggregation at time t as:

$$r_n^t = \begin{cases} f(x_0^t, f(x_1^t, \dots f(x_{N_t}^t, i_n^t)\dots)) & \iff N_t > 0 \\ i_t & \iff N_t = 0 \end{cases}$$

The formula describes a universal and common, in many functional languages, way of ***folding*** a list of elements to a value of a particular type. Different effects can be achieved when a particular aggregation function and the initialisation value are chosen. For example, by choosing the standard addition operator as the aggregation function and 0 as the initial value, all received values can be summed up together. Alternatively, the maximum of the received values can be calculated when the max aggregation function and 0 as the initialisation value are used. Note that it is possible that the aggregated and the initialisation streams have different data types as long as the aggregation function has the following general signature: $f : X \times Y \to Y$.

SOSNA uses the **aggr** operator to specify neighbourhood stream aggregation. The operator takes four arguments which, in addition to the above defined three aggregation components, include a specification of an abstract *network topology*. For simplicity, we confine ourselves in this presentation exclusively to neighbourhood topologies and defer further discussion to the later part of this section. The following stream function defines a universal spatial averaging procedure that takes as arguments the stream whose values are to be averaged and the topology along which the averaging is to be performed. The function is applied to the stream of sensor readings and to a one-hop neighbourhood topology. The aggregation process operates on tuples that comprise the sum of sensor readings and the number of values comprising the sum. The final result is a simple division of the sum by the number of data items.

```
(def (avg stream topology)
  (let (x (tuple stream 1))
    (let ((tuple sum n) (aggr vector_add x x topology))
      (/ sum n))))
(output (Display (avg (vals sensor) (nhood nodes))))
```

Note that the initialisation and the aggregated streams are the same. This is because the aggregation procedure does not include the the value the aggregated stream takes (if present) at the receiving nodes. These values, however, can be added either after the aggregate value has been computed or they can be smuggled in as the initial value. Table 3.6 presents an example evaluation of the above

Stream	Values							
				node A				
(**vals** sensor)	115	118	117	116	115	114	113	...
n	3	3	3	3	3	3	3	...
sum	348	351	345	342	345	342	339	...
Display	116	117	115	114	115	114	113	...
				node B				
(**vals** sensor)	117	116	115	114	113	112	110	...
n	2	2	2	2	2	2	2	...
sum	232	234	232	230	229	226	223	...
Display	116	117	116	115	114	113	111	...
				node C				
(**vals** sensor)	116	117	113	112	116	116	116	...
n	2	2	2	2	2	2	2	...
sum	231	235	230	228	232	230	229	...
Display	115	117	115	114	116	115	114	...

Table 3.6: Example evaluation of the neighbour stream averaging function (integer calculations).

function when applied to a stream of sensor values. We use the physical network topology presented in Figure 3.7 and, additionally, assume that there are no communication failures.

The main reason we used a single aggregation process operating on a stream of tuples instead of two independent **aggr** operators is such that SOSNA does not guarantee data consistency across different aggregation operations. In other words, two different instances of the same aggregation process may return different results given a particular pattern of lost messages. Although the current SOSNA implementation schedules independent **aggr** operators to be executed in parallel by means of using the same radio message for their data items, reliance on this property would make reasoning about the properties of larger programs difficult, as well as, it would preclude the implementation of alternative communication scheduling policies.

Neighbourhood stream aggregation can be used in the recursive setting to implement, for example, persistent state that is robust to different node start-up times. In the following example nodes count to infinity and synchronise the counter value with their neighbours.

```
(def-rec sync_n
  (let (prev_n (fby 0 sync_n))
    (+ 1 (aggr min prev_n prev_n (nhood nodes)))))
```

In every instant, nodes synchronise their counters to the lowest value of those held by their one-hop neighbours. Since all nodes increment their counters, a given node's counter value can decrease only as long as there exist network nodes with smaller counter values. As a result, when this program is run in a multi-hop network with arbitrary node start-up times, synchronization of counting gradually spreads across the entire network leading to global convergence to the lowest counter value. The above program is an example of an *amorphous algorithm* that employs local communication for periodic spreading of *network gradients* in order to realise a spatial computational process that is robust to network failures and is independent of the underlying network topology. Beal & Bachrach (2006) show how such algorithms can be implemented in the Proto language using neighbourhood stream aggregation.

Contrary to Proto, SOSNA allows more sophisticated communication patterns to take place within one instant. Consider, for example, the case in which one aggregation expression depends on the result of another. Clearly, the corresponding data aggregation rounds cannot be scheduled for execution in parallel and hence must be executed one after another in the same instant. This property is the core characteristic of the language - instants may comprise complex communication patterns so that applications can realise the whole sensor data transformation path in bounded time. For example, the following stream definition realises three hop-wide sensor value averaging within one instant:

```
(def (avg1hop stream)
  (avg stream (nhood nodes)))
(def avg3hop (avg1hop (avg1hop (avg1hop (vals sensor)))))
```

Multi-hop Stream Aggregation

In addition to the basic neighbourhood stream aggregation, the **aggr** operator can be used to perform multi-hop stream aggregation when the **nhood** expression is replaced with a multi-hop topology expression. SOSNA uses network spanning trees to direct the aggregation of stream values spread across larger network regions. Multi-hop stream aggregation is semantically similar to neighbourhood stream aggregation and it is also an atomic operation that is executed within one instant.

Data aggregation along multi-hop tree topologies is one of the most fundamental communication paradigms for wireless sensor networks. It allows to collect sensor data from multiple sensor nodes and conserve energy by fusing different data packets at tree nodes instead of forwarding every one of them to the root of the tree, which is often referred to as the *sink*. In the context of macro-programming languages, tree-based data aggregation is often used implicitly as means to implement high-level data transformation primitives. For example, *Madden et al. (2005)* use tree-based periodic data aggregation as the underlying topology for dissemination and execution of their SQL-like queries while Newton et al. (2007) use a similar communication paradigm to implement abstract region data aggregation operators. In tree-based data aggregation, data flows from the nodes that are at the bottom of the tree towards their parents, i.e., their neighbouring nodes that are one-hop closer to the

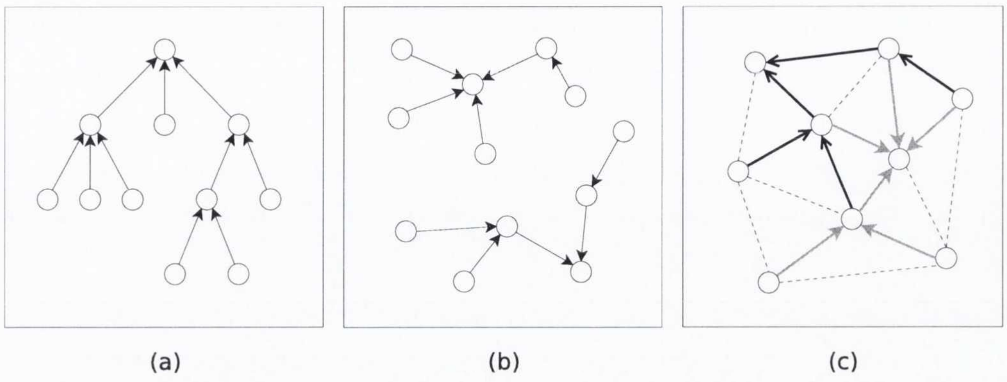


Figure 3.8: The flow of information in tree-based data aggregation (a), cluster topology comprising two trees (b), two overlapping cluster topologies defined in the same program (c).

sink node. The parent nodes fuse the received values to a single one and forward it one level up so that eventually the process ends at the root of the tree. Figure 3.8 (a) describes the flow of aggregated data along an example tree topology.

Cluster topology is a union of several disjoint network spanning trees. SOSNA defines cluster topologies in which all trees have bounded depth and uses them to direct multi-hop stream aggregation. Cluster topologies can be defined in the language either statically, or dynamically, based on the changing values of a given stream. SOSNA programs can define multiple, possibly overlapping cluster topologies, hence, individual nodes may take different roles with respect to different topologies. The idea is described in Figure 3.8(c) in which two different cluster topologies are defined over the same *physical* network topology. The arrows represent the parent/child relation among the topology members while the dashed lines correspond to the physical wireless connectivity of the network. We use the term *cluster heads* to refer to the collection of network nodes that are the roots of all spanning trees comprising a given cluster topology and the term *cluster members* to refer to all nodes that are part of that topology. Multi-hop stream aggregation is defined in exactly the same way for both static and dynamic cluster topologies. Hence, in this section we focus only on dynamic cluster topologies and leave the description of static cluster topologies to Section 3.2.3.

Dynamic cluster topologies are constructed using a bounded-hop leader election protocol that selects the topology's root nodes based on the values of a *selection stream*. Once the root nodes are chosen, all nodes participating in the process establish their roles within the topology based on the communication distance towards the closest root node. The process is repeated in each instant in order to reflect possible changes in the values of the selection stream. The election process comprises internally a fixed number of data aggregation rounds in which the participating nodes exchange their current values of the selection stream, forward the best leader value received so far, and keep track of the network identifiers of those nodes from which the best value was received so that the topology can be established in parallel to the leader election process. The number of aggregation rounds comprising the process defines the maximum depth of the resulting topology and is a compilation parameter that can be tuned to reflect application-specific requirements. The details of the protocol are presented in Chapter 4.

There are two ways of choosing the root nodes based on the values of the selection stream: we can pick either the locally largest value or the locally smallest one. SOSNA uses, therefore, two stream operators **clmax** and **clmin** to construct dynamic cluster topologies based on the values of a given selection stream. The following program fragment aggregates sensor readings along a dynamic cluster topology in which the root nodes hold the largest sensor reading values in their network vicinity.

```
(def val (vals sensor))
(def sum (aggr + val val (clmax val)))
```

Note that, similarly to neighbourhood stream aggregation, the local value of the aggregated stream at the receiving nodes is not included in the aggregation result unless explicitly added (e.g., as the initialisation stream). Also, the receiving nodes, because of the uni-directional nature of data aggregation over trees, are always the topology's cluster heads, hence, the result of aggregation is absent on all cluster members.

Assuming that the `val` stream is always present on all network nodes, the topology defined by this program will cover the whole network resulting in all nodes taking part in the topology construction and aggregation processes. However, applications such as target tracking may require that the extent of communications is confined to a relatively small network region surrounding, for example, the physical object being tracked. Such an effect can be achieved in SOSNA through spatial filtering of the selection stream because the extent of a dynamic topology is also limited by the extent of the spatial region associated with the values of the selection stream. For example, assuming that the relevant sensor readings are those above 100, the following modification can be used to achieve the effect.

```
(def val (within (> (vals sensor) 100)
                 (vals sensor)))
```

As a result, the topology is created only around the tracked object as long as it is within the network's sensing range. The topology's cluster heads are likely to be the nodes located the closest to the object since they are the ones that receive the strongest signal. Also, as the object moves, the topology's structure is reevaluated to reflect the change. Figure 3.9 depicts three consecutive instants of the topology's evolution.

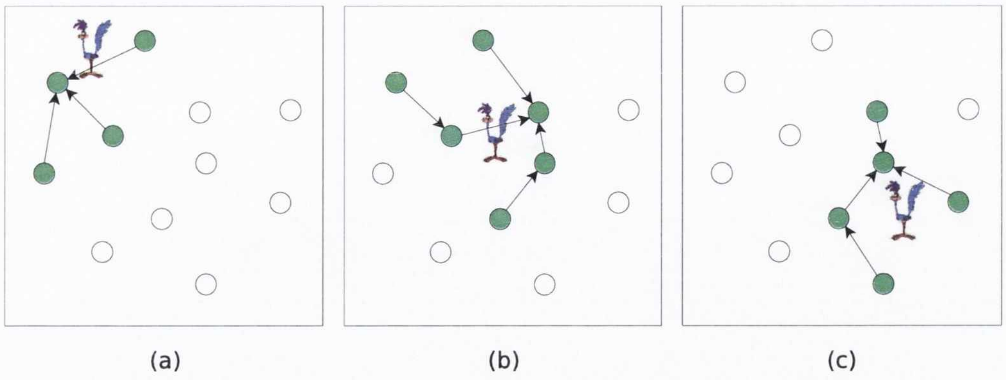


Figure 3.9: The evolution of a dynamic topology in a simple target tracking application.

Multi-hop stream aggregation is an atomic operation in the sense that it is executed entirely within one instant. However, contrary to neighbourhood stream aggregation, the process internally comprises several rounds of neighbourhood data aggregation that are executed sequentially. We use the term *aggregation steps* to refer to the individual data aggregation rounds comprising the whole multi-hop aggregation process. The exact number of aggregation steps is a compilation parameter and is equal to the maximum depth of all spanning trees comprising a given cluster topology. Multi-hop stream aggregation has exactly the same syntax as neighbourhood stream aggregation: the **aggr** operator requires specification of the aggregation function, the initialisation stream, the aggregated stream and the cluster topology to be used to guide the process:

```
(aggr F I S C)
```

The semantics of multi-hop stream aggregation along a cluster topology `C` of depth $d > 0$ are the following. In every instant, there are d aggregation steps. In each step $k = 0, \ldots, d-1$ nodes that are at the level (depth) $d-k$ in their corresponding spanning trees send a data value v_k to their parent nodes (the receiving nodes) that are at the level $d-k-1$. The receiving nodes fuse the received values in the same way as during neighbourhood stream aggregation using the aggregation function `F` and an initialisation value i_k. In order to facilitate the presentation, we introduce the `aggrv` operator whose semantics differ from the semantics of the **aggr** operator in that the operator deals with data values rather than streams.

The aggregation step sequence can be viewed as a composition of the following sequence of `aggrv` operator instances, for $k = 0, \ldots, d-1$:

$$v_{k+1} = (\texttt{aggrv} \quad F \quad i_k \quad v_k \quad c_k),$$

where $v_0 = S^t$ and

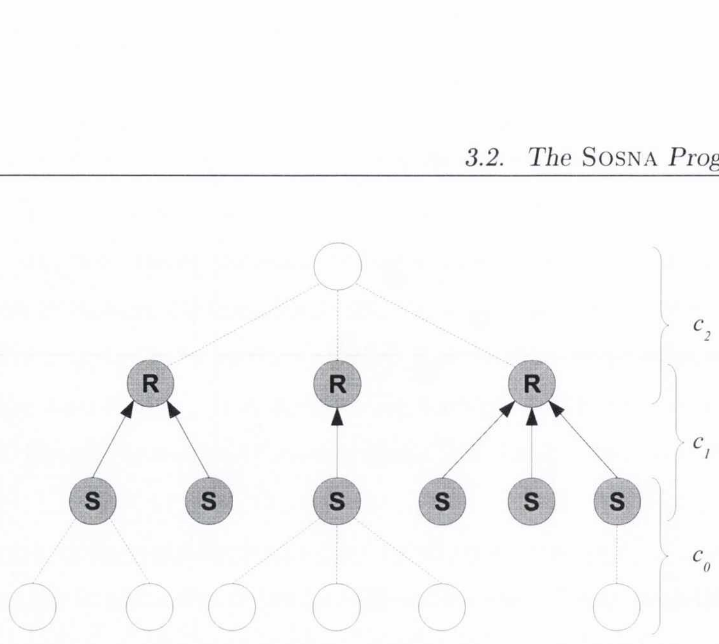


Figure 3.10: Example neighbourhood aggregation topology c_1 defined for the second ($k = 1$) aggregation step in a spanning tree of depth $d = 3$.

$$i_k = \begin{cases} S^t & \iff \quad k = 0, \dots, d-2 \wedge d > 1 \\ I^t & \iff \quad k = d-1 \end{cases}$$

where S^t is the current value of the aggregated stream `S`, I^t is the current value of the initialisation stream `I`, and where c_k are neighbourhood topologies defined as follows. Each c_k comprises those network nodes that belong to `C` and that are at the levels $d - k$ and $d - k - 1$ in their corresponding spanning trees. Of those nodes, the first group are the senders of the aggregated data values and the second group comprises the receivers of the values sent by the nodes from the first group (see Figure 3.10).

Following the semantics of neighbourhood stream aggregation, for each $k = 0, \dots, d - 1$ the result of the corresponding multi-hop aggregation step is present on the receivers of c_k. Therefore, the final result v_d of the process will be present on the receivers of c_{d-1}, i.e., the `C`'s cluster heads. Also, because the values of the initialisation stream `I` are used only in the last aggregation step, their presence defines the presence of the result of the whole multi-hop aggregation process. Note that, since the remaining aggregation steps use the aggregated stream's values as the initialisation value, both `I` and `S` streams have to be of the same type. Additionally, we can see that if we pick $d = 1$ and substitute a general neighbourhood topology for c_0 the above definitions correspond the basic neighbourhood stream aggregation, as described in the previous section.

Stream propagation

In multi-hop stream aggregation data flows from cluster members to cluster heads and the result of the operation is present on cluster heads. Some WSAN applications may require, however, that the data flows in the opposite direction. Consider, for example, an application in which a controller needs to instruct sensor nodes as to the values of certain parameters such as, for example, sensor sensitivity or an adaptive detection threshold.

Stream propagation is an operation whose execution is guided by network topologies in the same way as in the case of stream aggregation but in which the direction of data flow is reversed. Both neighbourhood and cluster topologies can be used to propagate data along them and, similarly to stream aggregation, the propagation process is semantically equivalent to a sequence of neighbourhood data aggregation steps. SOSNA uses the **expand** operator to express stream propagation and the operator takes the same arguments as the **aggr** operator: a *propagation function*, an initialisation stream, a *propagated stream*, and a topology that is used to guide the whole process:

```
(expand F I S C)
```

In stream propagation the sending nodes, i.e., those for which the propagated stream takes non-null values and which, in the case of cluster topologies, are also cluster heads, send their current values of the propagated stream to all other members of the topology. The recipients of these data values use the propagation function and their local values of the initialisation stream to compute the local result of the propagation process and, in the case of cluster topologies, they forward this result further down their spanning trees.

Stream propagation is a form of proactive data sharing and can be useful in applications that require actuator coordination. For example, the following program uses multi-hop stream propagation in order to share the maximum value of the relevant sensor readings among sensor/actuator nodes so that they can decide individually on whether to actuate or not:

```
(def d (within (> (vals sensor) 100)
          (vals sensor)))
(def top (clmax d))
(def avg_d (avg d top))
(def-fun (copy x y) x)
(def diff (- d (expand copy 0 avg_d top)))
(output (actuate (within (> diff 10) true)))
```

The program first spatially filters all sensor readings retaining those that are greater than 100, then a dynamic cluster topology is defined around nodes with maximal detection values so that the spatial average of the detection values can be computed (`avg_d`). We use the same, general averaging function that was defined in Section 3.2.2 but this time parametrising it with a multi-hop topology. The average value is then propagated back to cluster members by simply copying it, i.e., without transforming it in any way, so that the difference between nodes' local sensor reading and the average reading value (`diff`) can be calculated and acted upon.

Homogeneous neighbourhood stream propagation (Section 3.2.3 discusses more general, heterogeneous topologies) is equivalent to homogeneous neighbourhood stream aggregation because these topologies do not distinguish any nodes and the sending/receiving roles are decided entirely based on the presence and absence patterns of the initialisation and the aggregated/propagated streams. Therefore, assuming no communication failures, the following expression will always evaluate to `true` for all device classes `n`:

```
(== (expand F I S (nhood n))
    (aggr F I S (nhood n)))
```

Multi-hop stream propagation can be seen essentially as reversed multi-hop stream aggregation. However, because the propagated data values flow down the topology's spanning trees the description can be simplified. Therefore, following closely the presentation of multi-hop stream aggregation, we define the semantics of multi-hop stream propagation along a cluster topology `C` of depth $d > 0$ as follows. In every instant, there are d propagation steps. In each step $k = 0, \ldots, d-1$, nodes that are at the level k in their corresponding spanning trees send a data value v_k to their tree children that are at the level $k+1$. The receiving nodes fuse the received values according to the rules of neighbourhood stream propagation using the propagation function `F` and an initialisation value i_k. Note, that because every child node has only one parent, each receiving node will receive at most one value.

Analogously to the previous section, we introduce the `expandv` operator in order to express a single-hop neighbourhood data propagation step in terms of individual data values. Therefore, the propagation step sequence is a composition of the following sequence of `expandv` operator instances for $k = 0, \ldots, d-1$:

$$v_{k+1} = (\texttt{expandv} \quad F \quad I^t \quad v_k \quad c_k)$$

where $v_0 = S^t$ is the current value of the propagated stream `S`, I^t represents the current value of the initialisation stream `I`, and where c_k are neighbourhood topologies defined as follows. Each c_k comprises those network nodes that belong to `C` and that are at the levels k or $k+1$ in their corresponding spanning trees. The nodes from the k^{th} level send the propagated data values and the nodes that are one level below them are the receivers of these values (see Figure 3.11 for an example propagation step topology definition).

Contrary to multi-hop stream aggregation, SOSNA defines the final result of the propagation process to be present on all cluster member nodes for which the initialisation stream takes non-null values and

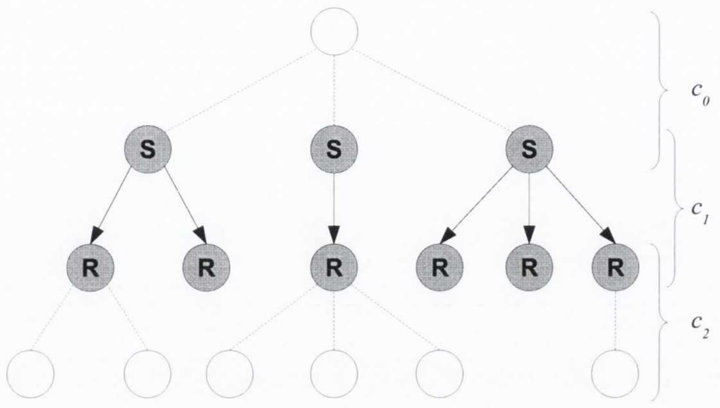


Figure 3.11: Example neighbourhood propagation topology c_1 defined for the second ($k = 1$) aggregation step in a spanning tree of depth $d = 3$.

to be absent on the cluster head nodes. Therefore, each of v_k for $k = 1, \ldots, d$ becomes part of the final result since these values are computed by different sets of nodes and, hence, it is possible, depending on the choice of the propagation function, that the process ends with the topology members taking distinct values as their final result. This property may be useful, for example, to realise a process in which data items are propagated among cluster members conditionally, based on the result of their local computations. Finally, similarly to multi-hop stream aggregation, both the initialisation and the propagated streams are required to be of the same data type.

3.2.3 Network Heterogeneity

Heterogeneity is a common characteristic of wireless sensor-actuator networks in which sensing, actuation and control might be performed by devices of different types and capabilities. SOSNA supports this variety through the concept of *device classes* that represent in programs different types of network nodes. The language does not relate device class definitions to any specific hardware platforms but rather treats them as abstract descriptors with application-specific semantics. Once defined, device classes are used to parametrise stream and topology definitions so that the global program can be automatically partitioned into a set of subprograms each of which to be executed on network nodes belonging to a different device class.

Device classes need to be explicitly specified for program inputs and static (as opposed to dynamic) topology definitions. SOSNA's type system allows the compiler to infer device class information for most of program streams but in cases when this is not possible the `at` operator can be used to hint the inference process. Consider the following program:

```
(def-class motes)
(def-input sensor @ motes : int)
(output (Display (/ (vals sensor) 2))
        (Report (at motes 1)))
```

Knowing that the `sensor` input is evaluated on devices from the `motes` class, the compiler infers that the `Display` actuator must be present on the same group of devices. In the second case, however, the device class of the constant stream `1` is unknown, hence, the stream needs to be explicitly annotated. Note, that the constant stream `2` is automatically inferred to be evaluated by the `motes` device class because it is referenced in the evaluation of an arithmetical stream expression involving a stream of a known device class.

SOSNA organises device class definitions hierarchically in order to provide a network-wide time synchronization model in which the top-level device class represents the global network time source. The model imposes a global spanning tree topology on the whole network and this tree directly corresponds to the global network time synchronization tree. This way, both the time synchronization and the static topology can be constructed at the same time without additional overhead. In order to find a balance between the determinism of static topology specification and the robustness to dynamic network topology changes, SOSNA programs define only the abstract relations among device classes leaving the organisation of network nodes *within* a device class up to the runtime system. Also, the language semantics do not specify which of the top-level device class nodes should be the time source for the whole network. Instead, runtime implementations can use different strategies to better fit the application scenario. For example, a single node can be dynamically chosen to be the global time source, or all nodes in the top-level device class can synchronise to an external NTP[2] time server.

The choice of using static device class hierarchies instead of a more dynamic setting, in which relations among different device classes are established at run time, carries the benefit of increased predictability of program operation, as well as it simplifies reasoning about the possible interactions among different network nodes. Constraining the range of possible network topologies in the program code helps to ensure timeliness of network communications since no longer general routing mechanisms need to be used. Many different WSAN applications fit in this hierarchical and static architecture, for example, building management systems or the CSOnet system described in Section 1.1. Nevertheless, the model has also its limitations making it difficult, in particular, to support applications that employ many mobile network nodes.

Since device classes represent groups of network nodes, hierarchical device class composition corre-

[2]Network Time Protocol.

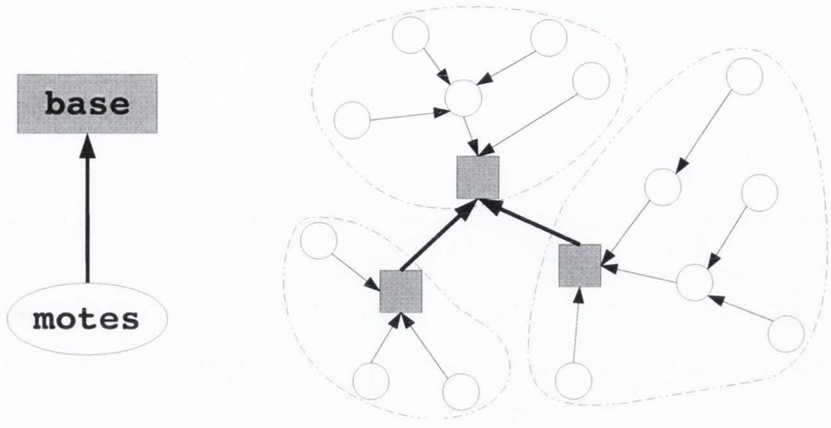


Figure 3.12: The relation between static clusters of an example device class hierarchy and its global network time synchronization spanning tree topology.

sponds to a certain partitioning of the global time synchronization tree. For each pair of device classes C_0 and C_1 such that C_1 is a subclass of C_0 we define ***static cluster*** to be the maximal sub-tree of the the time synchronization tree that is rooted at a network node $n_0 \in C_0$ and comprising a number of network nodes $n \in C_1$. For the same pair of device classes C_0 and C_1 we also define ***static cluster topology*** to be a sum of all static clusters defined on C_0 and C_1. Additionally, we use the terms *major device class* and *minor device class*, to refer to C_0 and C_1 respectively. Figure 3.12 demonstrates the relation between a hierarchy of two device classes and its corresponding static clusters defined on an example network time synchronization topology.

Static cluster topologies can be used to aggregate and propagate stream values in the same way as dynamic cluster topologies do. The **`clstat`** operator is used to define static cluster topologies and it takes one argument that is the name of a minor device class (the name of the major device class is inferred from the device class hierarchy). The following program defines the device class hierarchy depicted in Figure 3.12 and it implements aggregation of sensor readings that originate only from the `motes` device class.

```
(def-class base)
(def-class mote : base)
(def-input sensor @ motes : int)
(output (Report (aggr + 0 (vals sensor) (clstat mote))))
```

If we assume the time synchronization topology presented in Figure 3.12, then this program implements multi-hop stream aggregation, which proceeds independently along the three static clusters and the results of which are present on the three cluster heads, i.e., nodes from the `base` device class. In accordance with the semantics of multi-hop aggregation, the initialisation stream `0` is evaluated within the `base` device class while the values of the aggregated stream come from the `mote` class.

Hierarchical device class organisation serves in SOSNA as an abstract and general network model. Because many different physical network topologies may conform to the same device class hierarchy (for example, different static clusters may have different depths or they may comprise different numbers of `base` nodes), the language's semantics do not assume anything more than the maximum depth of all static clusters. Since this value may be application-specific, SOSNA treats it, similarly to the case of dynamic cluster topologies, as a compilation parameter. It is only a language design choice, however, that all static cluster topologies must have the same maximum depth because the synchronous macro-programming model requires only that the extent of all communications within one instant is bounded. Hence different device classes could be specified maximal cluster depths on an individual basis.

Static clusters are used to guide information flow along the arcs of the device class hierarchy. SOSNA allows also for one-hop communication among nodes from different device classes, as long as, they are subclasses of the same device class. Consider the following program in which actuators act based on the maximal values of detection reports provided by nearby sensor nodes.

```
(def-class base)
(def-class mote : base)
(def-class actuator : base)
(def-input light @ motes : int)
(def detection (within (> (vals sensor) 100)
                       (vals sensor)))
(output (act (aggr max 0 detection (onehop actuator mote))))
```

The **onehop** operator is used to define heterogeneous neighbourhood topologies. Stream aggregation along these topologies is performed in much the same way as in the case of homogeneous neighbourhood stream aggregation but with the difference that the sending and the receiving nodes belong to different device classes. The operator's first argument specifies the device class of the receiving nodes while the second one defines the device class of the sending nodes.

Compilation of this program will result in three different node-local programs being generated for execution on devices from each of the three device classes. Because the `base` device class is used only as means to define the other two device classes and none of the program streams are evaluated by nodes of this class, the compiler generates for it only a minimal program whose task is act as the time source for the whole network. Therefore, deploying this application on a real network will require installation, in addition to sensor and actuator nodes, a single one node executing the code generated for the `base` device class so that it can enable execution of other nodes. In contrast to the `base` program, the `mote` program will include a driver for the `sensor` input, the code for evaluation of the `detection` stream and the data sending code for the aggregation operator. The `actuator` program will include a driver for the `act` actuator, data reception and fusion code for the aggregation operator and the code for the `max` function.

The automatic program partitioning technique employed by SOSNA rests on the general idea that only the semantics of communication operators, i.e., stream aggregation and propagation, can span different device classes allowing for a robust implementation because the operators can handle communication failures in a controlled and concise manner. Section 3.3 describes SOSNA's type system and discusses the inference process in more detail.

3.3 The Type System

SOSNA implements the classical ML-style type system (Pierce 2004) that is a type discipline based on the simply-typed λ-calculus ($\lambda_{\rightarrow}$) and extended with polymorphic `let` declarations. λ-calculus is a primitive but expressive formalism of functions and values used as the basis for most of functional programming languages. The simply-typed λ-calculus is a typed interpretation of the λ-calculus that uses, in its most basic form, only one type constructor $\rightarrow$ to denote function types. `let` declarations (see Section 3.2.1 of an overview of how they are used in SOSNA) extend $\lambda_{\rightarrow}$ with polymorphic types, i.e., generic type definitions comprising type variables that may be substituted with different type expressions to form new type *instances*. We follow the terminology introduced in Chapter 10 of Pierce (2004) where the authors refer to the core language underlying the ML type discipline as *ML-the-calculus* and to the type system itself as *ML-the-type-system*. In the following sections, we introduce the basic concepts of the ML type discipline that forms the core of the SOSNA's type system. Then, we present the overall SOSNA's type classification and discuss the typing of the language's core constructs.

3.3.1 The ML Type Discipline

ML-the-calculus[3] is a generic language that, in its simplest form, comprises constants (language literals), identifiers (program variables), functions, function application expressions and local **let** definitions. The language can be presented by the following grammar:

$$e \ ::= \ c \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e$$

[3]We present only a simplified description of the language and its type system. For a comprehensive presentation see Pierce (2004), Chapter 10.

Types in ML-the-type-system are built from type constructors and type variables. The former may have an arbitrary number of arguments that can either be types or type variables and the latter are intended as a way of expressing partially-defined types. Type constructors that have zero arguments are called primitive types and can be used for the definition of such basic types as integer numbers or strings. The most fundamental type constructor in ML-the-type-system is the function type constructor $\rightarrow$ that, by default, takes two arguments that correspond to the type of a function's argument and to the type of its return values. As a result, the set of all acceptable types can be presented in the form of a grammar of type constructors and type variables:

$$\mathsf{T} \quad ::= \quad \mathsf{X} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \ldots$$

The key characteristic of the ML type discipline is the ability to assign polymorphic types to **let**-bound program variables. Consider, for example, a function that computes the length of a list. Clearly, this abstract computational task can be formulated independently of the type of the list's elements and yet, if the function were to receive a monomorphic type, a different version would have to be implemented each time a list of elements of a different data type is used.

The **let**-bound identifiers are assigned generic types called *type schemes*. Type schemes have a general form $\forall\bar{\mathsf{X}}.\mathsf{T}$ where $\bar{\mathsf{X}}$ represents a set of type variables that are considered bound within T. When **let**-bound identifiers are referenced in an expression, their corresponding type scheme variables are instantiated with concrete types (which in turn might comprise other variables), to form the identifier's type *instance*. The resulting type is denoted as $[\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}$ and represents a type substitution that maps variables from $\bar{\mathsf{X}}$ to types from $\bar{\mathsf{T}}$ in the type scheme $\forall\bar{\mathsf{X}}.\mathsf{T}$. For example, the length-computing function could be assigned the following type scheme:

$$\texttt{length} : \forall \mathsf{X}.\mathsf{List}(\mathsf{X}) \rightarrow \mathsf{Int}$$

and when applied to a list of Boolean values, this instance would receive a concrete type

$$[\mathsf{X} \mapsto \mathsf{Bool}]\mathsf{List}(\mathsf{X}) \rightarrow \mathsf{Int} \equiv \mathsf{List}(\mathsf{Bool}) \rightarrow \mathsf{Int}.$$

In order to present how ML-the-calculus' programs are typed of we need to first introduce several definitions. A *typing assumption* $x : \tau$ is a statement that says that the program identifier x has type τ. A *typing context* Γ is a set of typing assumptions. The *typing relation* $\Gamma \vdash e : \tau$ states that the expression e has the type τ in the typing context Γ. Sometimes we want to extend a typing context with an additional typing assumption about an identifier to form a new context in which case we write $\Gamma; x : \tau_0 \vdash e : \tau_1$. Also, when the typing context is empty, we simply write $\vdash e : \tau$ meaning that e has type τ without additional typing assumptions.

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}(\textsc{ml-Var}) \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma; x : \sigma \vdash e_2 : \mathsf{T}}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mathsf{T}}(\textsc{ml-Let})$$

$$\frac{\Gamma; x : \mathsf{T} \vdash e : \mathsf{T}'}{\Gamma \vdash \lambda x.e : \mathsf{T} \to \mathsf{T}'}(\textsc{ml-Abs}) \qquad \frac{\Gamma \vdash e : \mathsf{T} \quad \bar{\mathsf{X}} \cap ftv(\Gamma) = \emptyset}{\Gamma \vdash e : \forall \bar{\mathsf{X}}.\mathsf{T}}(\textsc{ml-Gen})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \to \mathsf{T}' \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash e_1\ e_2 : \mathsf{T}'}(\textsc{ml-App}) \qquad \frac{\Gamma \vdash e : \forall \bar{\mathsf{X}}.\mathsf{T}}{\Gamma \vdash e : [\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]\mathsf{T}}(\textsc{ml-Inst})$$

Figure 3.13: Typing rules for ML-the-type-system.

Figure 3.13 presents a set of *typing rules* that define ML-the-type-system. The upper part of each of these rules is called the premise and the bottom parts are referred to as conclusions. The typing rules should be read as follows: if the premise holds then the conclusion can be drawn. Therefore, the ML-VAR rule says that if the typing context Γ contains a type scheme definition for the identifier x, then the variable reference expression x receives the type (scheme) σ. The ML-ABS rule establishes the typing of single-argument function definitions: if the function body expression receives type T' under the assumption that the function's argument x has type T, then the function receives type $\mathsf{T} \to \mathsf{T}'$. The ML-APP rule defines the typing of function application and its definition is straightforward: if the function expression e_1 is assigned a function type and the argument expression e_2 is assigned the type of the function's argument, then the result of applying the function is the same as the function's return type.

The ML-LET rule defines the typing of `let` expressions. Note that, contrary to the ML-APP rule, the identifier x is assumed to have a generic type and hence the expression e_1 is required to type-check as a type scheme. It is exactly the difference in treatment of functions and `let` expressions, i.e., that function arguments receive concrete types while `let`-bound identifiers receive generic types, that makes type inference for ML-the-calculus decidable, as opposed to more general formalisms (see Pierce 2004, p. 390).

The ML-GEN and ML-INST rules are specific to ML-the-type-system and together with the notion of type schemes they distinguish the formalism from the simply-typed λ-calculus. The rules are used respectively to introduce type schemes when they are needed (type generalisation) and to instantiate them when a concrete type is required. The ML-GEN rule converts a concrete type T into a type scheme by quantifying over some its type variables. The choice of these variables can be arbitrary, as long as, these variables are not free in Γ (a type variable X is free with respect to a type scheme $\forall\bar{\mathsf{X}}.\mathsf{T}$ when it occurs in T and $\mathsf{X} \notin \bar{\mathsf{X}}$). Since the rules do not specify which variables need to be generalised

$$
\begin{array}{lcl}
\mathsf{T} & ::= & \mathsf{Bool} \mid \mathsf{Byte} \mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Time} \mid \mathsf{T} \times \mathsf{T} \mid \mathsf{X_T} \mid \ldots \\
\mathsf{S} & ::= & \mathsf{Stream(T, C)} \mid \mathsf{Input(T, C)} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \mathsf{S} \Rightarrow \mathsf{S} \mid \\
 & & \mathsf{Topology(K, C, C)} \mid \mathsf{DevClassId(C)} \mid \mathsf{Unit} \mid \mathsf{X_S} \\
\mathsf{C} & ::= & \mathsf{TimeSrc} \mid \mathsf{Subclass(C, N)} \mid \mathsf{X_C} \\
\mathsf{K} & ::= & \mathsf{NHood} \mid \mathsf{CStatic} \mid \mathsf{CDynamic} \mid \mathsf{X_K} \\
\mathsf{N} & ::= & \mathsf{X_N} \mid \mathsf{n_0} \mid \mathsf{n_1} \mid \ldots
\end{array}
$$

Figure 3.14: The grammar of SOSNA's types.

nor what types should be substituted for type variables during type scheme instantiation they are not particularly useful for building a real type checker. Instead, they define the general steps any ML type checker would have to make in order to analyse a program while leaving the details of decision making unspecified.

3.3.2 Classification of SOSNA's Types

SOSNA divides all program types into two main categories: scalar domain types and stream domain types. The former category contains all scalar data types such as, for example, integer or floating point numbers while the latter comprises the types of all other language constructs. The language declares also several auxiliary type classes but they cannot be directly ascribed to program expressions. SOSNA's types, therefore, conform to the type grammar presented in Figure 3.14.

For consistency reasons, each of the above type categories defines its own set of type variables that range only over types from this category. The first type category comprises scalar domain types and, besides primitive data types, it includes also tuple types. It is assumed that future versions of the language can extend this group with more versatile types such as, for example, arrays. The stream domain type category S comprises types of the core language constructs and they are respectively: streams, inputs, scalar functions, stream functions, topologies, device class identifiers, and the program output.

The auxiliary C type category represents device class types. This is in contrast to the device class identifiers that represent abstract device classes *in the language*, as opposed to the type system. In the rest of this presentation we will use the term (abstract) device class only within the realm of language semantics while we reserve the terms device class identifiers and device class types to represent abstract device classes within the language and within the type system respectively.

The grammar of device class types directly corresponds to the hierarchical organisation of abstract device classes: each device class is a subclass of another class. Device class types additionally include device class *names* that comprise the N type category and which directly correspond to device class program identifiers serving as a way of distinguishing among different subclasses of the same class.

The auxiliary K type category represents different types of SOSNA's topologies. These types are used as sub components the Topology type together with two device class types that correspond to the device classes of a topology's sending and receiving nodes. The category includes types representing neighbourhood, static and dynamic cluster topologies respectively.

Streams and Scalar Values

Because SOSNA's type system is split into two domains, i.e., the scalar domain and the stream domain, some of the language's syntactical constructs receive their types based on the *syntactical context* in which they appear. Literals such as, for example, numerical constants, can refer either to constant streams or to constant scalar values. The rule of thumb that is used to disambiguate these cases is such that whenever an expression e is a part of a surrounding function body definition, its type belongs to the scalar domain. We use, therefore, two different typing relations $\vdash_f$ and $\vdash_s$ to represent this distinction.

Constant streams in heterogeneous programs may be evaluated on devices of different classes because their semantics are universal. In order to represent this fact in the type system, SOSNA assigns polymorphic types to constant streams: each constant stream has a fixed data type but its device class is left unspecified until the context of the stream's instance is established. We can summarise the rules governing the typing of integer literals as follows:

$$\vdash_f 1 : \mathsf{Int}$$
$$\vdash_s 1 : \forall \mathsf{X_C}.\mathsf{Stream}(\mathsf{Int}, \mathsf{X_C})$$

The type inference algorithm infers correct type variable substitutions as it builds up the knowledge of relations among different type variables and all program expressions must receive concrete types after the algorithm finishes. It may happen, however, that there is not sufficient information to figure out all substitutions in which case the programmer needs to provide additional typing hints using, for example, the `at` operator, as described in Section 3.2.3.

SOSNA's type system defines streams in terms of the data type of their values and the class of devices they should be evaluated on. Contrary to other synchronous programming languages such as, for example, Lustre, SOSNA does not allow for compile-time analysis of stream value presence/absence patterns *throughout time*. This is motivated by the fact that it is impossible to predict the results of stream aggregation/propagation due to the possible communication failures. Also, the use of region-based computations gives rise to stream definitions that, in every instant, may take null values on a fraction of network nodes while non-null values on the remaining nodes. Finally, the unpredictability of the values of program inputs (which, in the case of detection sensors, can also be null) further increases the complexity. As a result, stream value presence has to be in many cases established at runtime. An alternative approach would be to extend the type system with additional annotations describing streams that *always* take non-null values (e.g, constant streams), and then to propagate this information throughout the program according to the semantics of the language's stream operators. However, due to the fact that such cases are likely to form a minor part of all program stream expressions, we decided to use runtime checks for stream value presence in all cases. Such an approach is a form of dynamic typing and popular programming languages such as, for example, Python apply it to the whole type system.

Inputs and Outputs

Contrary to constant streams, input declarations receive concrete types. For example, the following definition

```
(def-input x @ c : int)
```

results in the typing context Γ being extended with the assumption $\mathtt{x} : \mathsf{Input}(\mathsf{Int}, \mathsf{C})$ where C stands for any type from the C category such that $\Gamma \vdash \mathtt{c} : \mathsf{DevClassId}(\mathsf{C})$. Input values and their timestamps are accessed in programs through the use of the **vals** and **time** operators and the following rules define the typing process:

$$\frac{\Gamma \vdash_s e : \mathsf{Input}(\mathsf{T}, \mathsf{C})}{\Gamma \vdash_s (\mathtt{vals}\ e) : \mathsf{Stream}(\mathsf{T}, \mathsf{C})}\ (\textsc{Vals})$$

$$\frac{\Gamma \vdash_s e : \mathsf{Input}(\mathsf{T}, \mathsf{C})}{\Gamma \vdash_s (\mathtt{time}\ e) : \mathsf{Stream}(\mathsf{Time}, \mathsf{C})}\ (\textsc{Time})$$

The above rules say nothing more than: if the expression e is of type $\mathsf{Input}(\mathsf{T}, \mathsf{C})$ then the results of the expressions `(vals e)` and `(time e)` will have types $\mathsf{Stream}(\mathsf{T}, \mathsf{C})$ and $\mathsf{Stream}(\mathsf{Time}, \mathsf{C})$ respectively. Both T and C can be substituted any type from their respective categories, as long as, the substitution is consistent with all uses of the symbol within the context of a single rule.

The Unit type is used to describe the result type of the **output** operator. The type is used as means to ensure that all SOSNA programs define the actuation expression and that this expression is evaluated at the end of each instant.

Device Classes

SOSNA assigns the top-level device class the type $\mathsf{Subclass}(\mathsf{TimeSrc}, \mathsf{N})$, where N stands for a device class name corresponding to the program identifier associated with this device class. For example, the following declaration

```
(def-class base)
```

will result in the typing context being extended with the assumption $\texttt{base} : \mathsf{Subclass}(\mathsf{TimeSrc}, \mathsf{Base})$. The $\mathsf{TimeSrc}$ type cannot be ascribed to any program device class because the type system needs to keep track of all device class names. Hence, the type is used only as means to annotate the top-level $\mathsf{Subclass}$ type and SOSNA's syntax enforces that there can be only one top-level class definition. Subclasses of the top-level device class are assigned nested $\mathsf{Subclass}$ types therefore are clearly distinguishable from the top-level device class type. The following example declaration:

```
(def-class mote : base)
```

extends the typing context with the following assumption:

$$\texttt{mote} : \mathsf{Subclass}(\mathsf{Subclass}(\mathsf{TimeSrc}, \mathsf{Base}), \mathsf{Mote})$$

The `at` operator can be used to enforce stream expressions with unknown device classes to be evaluated within a given device class. Such a type-enforcing semantics is attributed by some functional languages to the ascription operator that operates on general-purpose data types. SOSNA, however, uses the `at` operator only with respect to device class types of stream expressions:

$$\frac{\Gamma \vdash_s e_0 : \mathsf{DevClassId}(\mathsf{C}) \qquad \Gamma \vdash_s e_1 : \mathsf{Stream}(\mathsf{T}, \mathsf{C})}{\Gamma \vdash_s (\texttt{at}\ e_0\ e_1) : \mathsf{Stream}(\mathsf{T}, \mathsf{C})}\,(\textsc{At})$$

Topologies

There are four different types of topologies that can be defined in SOSNA: homogeneous and heterogeneous neighbourhood topologies, static and dynamic cluster topologies and the $\mathsf{Topology}$ type is used to express all of these possibilities. The following rules define the typing of the neighbourhood topology constructors:

$$\frac{\Gamma \vdash_s e : \mathsf{DevClassId}(\mathsf{C})}{\Gamma \vdash_s (\texttt{nhood}\ e) : \mathsf{Topology}(\mathsf{NHood}, \mathsf{C}, \mathsf{C})}\,(\textsc{NHood})$$

$$\frac{\Gamma \vdash_s e_0 : \mathsf{DevClassId}(\mathsf{Subclass}(\mathsf{C}, \mathsf{N_0})) \qquad \Gamma \vdash_s e_1 : \mathsf{DevClassId}(\mathsf{Subclass}(\mathsf{C}, \mathsf{N_1}))}{\Gamma \vdash_s (\texttt{onehop}\ e_0\ e_1) : \mathsf{Topology}(\mathsf{NHood}, \mathsf{Subclass}(\mathsf{C}, \mathsf{N_0}), \mathsf{Subclass}(\mathsf{C}, \mathsf{N_1}))}\,(\textsc{OneHop})$$

As we can see, the Topology type carries two device class subtypes: the first one corresponds to the device class of topology's receiving nodes while the second one represents the device class of the sending nodes. We can see that homogeneous topologies are just a special case of heterogeneous topologies. In the case of homogeneous topologies, both device classes are the same meaning that the sending/receiving roles are established entirely at runtime. Similar rules govern typing of the cluster topology constructors and we present only the case of the **`clmax`** operator because typing rules for the **`clmin`** operator are analogous. Note, that in its current formulation, the language allows only integer streams to be used to define dynamic cluster topologies.

$$\frac{\Gamma \vdash_s e : \mathsf{Stream}(\mathsf{Int}, \mathsf{C})}{\Gamma \vdash_s (\texttt{clmax}\ e) : \mathsf{Topology}(\mathsf{CDynamic}, \mathsf{C}, \mathsf{C})}\,(\textsc{ClMax})$$

$$\frac{\Gamma \vdash_s e : \mathsf{DevClassId}(\mathsf{Subclass}(\mathsf{C}, \mathsf{N}))}{\Gamma \vdash_s (\texttt{clstat}\ e) : \mathsf{Topology}(\mathsf{CDynamic}, \mathsf{C}, \mathsf{Subclass}(\mathsf{C}, \mathsf{N}))}\,(\textsc{ClStat})$$

Scalar Functions

SOSNA constrains scalar function to be first-order. This means, that scalar functions may not take other functions as arguments or return them as values. Indeed, the language's type system allows scalar functions to operate only on types from the scalar domain which does not include the scalar functions type operator $\rightarrow$. Another consequence of such definition is that it is not possible in SOSNA to define a stream of functions or to refer to stream domain definitions from within a scalar function's body (with the exception of other scalar functions). For simplicity of this presentation, we confine the presentation to the case of single argument functions but the definitions can be easily generalised to the multi-argument case. The following rule defines the typing of anonymous scalar functions (scalar abstraction).

$$\frac{\Gamma; x : \mathsf{T} \vdash_f e : \mathsf{T}'}{\Gamma \vdash_s (\texttt{lambda}\ (x)\ e) : \mathsf{T} \rightarrow \mathsf{T}'}\,(\textsc{F-Abs})$$

This rule says simply that if the function body has type T' under the $\vdash_f$ typing relation and under the assumption that the function's argument has type T, then the function has type $T \rightarrow T'$. This formulation is practically the same as the one given by the ML-APP rule and the only difference is that two different typing relations are used in order to separate the scalar and stream domains.

The two-domain approach to typing requires a similar modification in the case of scalar function application as defined by the ML-APP rule of ML-the-type-system.

$$\frac{\Gamma \vdash_s e_1 : \mathsf{T} \rightarrow \mathsf{T}' \qquad \Gamma \vdash_f e_2 : \mathsf{T}}{\Gamma \vdash_f (e_1 \;\; e_2) : \mathsf{T}'}\text{(F-APP)}$$

Scalar functions, as we showed in Section 3.2.1, can be applied to non-null stream values point-wise in time. The following rule defines the typing of the **smap** operator (single-argument case).

$$\frac{\Gamma \vdash_s e_1 : \mathsf{T} \rightarrow \mathsf{T}' \qquad \Gamma \vdash_s e_2 : \mathsf{Stream}(\mathsf{T}, \mathsf{C})}{\Gamma \vdash_s (\texttt{smap}\ e_1\ e_2) : \mathsf{Stream}(\mathsf{T}', \mathsf{C})}\text{(SMAP)}$$

The **smap** operator when applied to a function of more than one argument enforces its both stream arguments to be evaluated on nodes of the same class. The following rule presents the typing of a two-argument **smap** operator. $(\mathsf{T}_1, \mathsf{T}_2) \rightarrow \mathsf{T}'$ represents the type of a two-argument scalar function.

$$\frac{\begin{array}{c}\Gamma \vdash_s e_f : (\mathsf{T}_1, \mathsf{T}_2) \rightarrow \mathsf{T}' \\ \Gamma \vdash_s e_1 : \mathsf{Stream}(\mathsf{T}_1, \mathsf{C}) \qquad \Gamma \vdash_s e_2 : \mathsf{Stream}(\mathsf{T}_2, \mathsf{C})\end{array}}{\Gamma \vdash_s (\texttt{smap}\ e_f\ e_1\ e_2) : \mathsf{Stream}(\mathsf{T}', \mathsf{C})}\text{(SMAP)}$$

Stream Functions

From the typing point of view, stream functions are equivalent to the ordinary functions of ML-the-calculus because they are higher-order, i.e., they can take other stream functions as arguments or return them as values, as well as, because they are typed in exactly the same way:

$$\frac{\Gamma, x : \mathsf{S} \vdash_s e : \mathsf{S}'}{\Gamma \vdash_s (\texttt{big-lambda}\ (x)\ e) : \mathsf{S} \Rightarrow \mathsf{S}'}\text{(S-ABS)}$$

Stream functions accept all stream domain constructs as arguments or as return values even though some of them such as, for example, device class identifiers have no runtime representation. This is, however, not an issue, because recursion is disallowed for stream functions and the compiler can, therefore, completely remove them from programs through in-lining.

3.3.3 Program Partitioning

SOSNA uses topologies to guide stream aggregation and propagation. The **aggr** and **expand** operators take a function, two streams and a topology as arguments and return a stream as a result. When the topology used is heterogeneous, the result stream is present on devices of a different class than the aggregated/propagated stream. The semantics of the aggregation and propagation operators spans, therefore, across different device classes and when the global program is partitioned these operators become the cut points of the process. The following rule defines the typing of the **aggr** operator (the **expand** operator is typed analogously)[4].

$$\frac{\begin{array}{ll}\Gamma \vdash_s e_f : (\mathsf{T}_2, \mathsf{T}_1) \to \mathsf{T}_1 & \Gamma \vdash_s e_1 : \mathsf{Stream}(\mathsf{T}_1, \mathsf{C}_1) \\ \Gamma \vdash_s e_t : \mathsf{Topology}(\mathsf{K}, \mathsf{C}_1, \mathsf{C}_2) & \Gamma \vdash_s e_2 : \mathsf{Stream}(\mathsf{T}_2, \mathsf{C}_2)\end{array}}{\Gamma \vdash_s (\mathsf{aggr}\ e_f\ e_1\ e_2\ e_t) : \mathsf{Stream}(\mathsf{T}_1, \mathsf{C}_1)}\ (\text{AGGR})$$

Having presented the SOSNA's type system in sufficient detail, we can give an example of how the compiler automatically partitions a global heterogeneous program into a set of homogeneous subprograms. Consider the following application:

```
(def-class base)
(def-class mote : base)
(def-class actuator : base)
(def-input sensor @ mote : int)
(def data (smap sqrt (vals sensor)))
(def result (aggr max 0 data (onehop actuator mote)))
(output (act result))
```

Assuming the typing context Γ to comprise the following assumptions:

`base`	: Subclass(TimeSrc, Base)
`mote`	: Subclass(Subclass(TimeSrc, Base), Mote)
`actuator`	: Subclass(Subclass(TimeSrc, Base), Actuator)
`sensor`	: Input(Int, Subclass(Subclass(TimeSrc, Base), Mote))
`sqrt`	: Int → Int
`max`	: (Int, Int) → Int

we can state that

$$\Gamma \vdash_s \texttt{data} : \mathsf{Stream}(\mathsf{Int}, \mathsf{Subclass}(\mathsf{Subclass}(\mathsf{TimeSrc}, \mathsf{Base}), \mathsf{Mote}))$$

and

$$\Gamma \vdash_s \texttt{result} : \mathsf{Stream}(\mathsf{Int}, \mathsf{Subclass}(\mathsf{Subclass}(\mathsf{TimeSrc}, \mathsf{Base}), \mathsf{Actuator})).$$

For each of the target device classes, the compiler has to filter the global program leaving only those stream and input definitions whose types indicate they should be evaluated on this device class. The **aggr** operator, since its semantics involve two different device classes (`mote` and `actuator`), is split

[4]The rule ignores, for simplicity, the additional type requirements for the aggregation function in case of aggregation over cluster topologies introduced in see Section 3.2.2.

into a sending and receiving parts that are embedded in the `mote` and `actuator` local programs respectively.

3.3.4 Time and Space Bounds

SOSNA programs execute in bounded memory and when a real-time flag is set, the compiler can generate code which is also guaranteed to execute in bounded time (provided that the network time synchronization protocol operates correctly). These properties are the consequences of the following properties of the language.

Scalar Domain

There are three main sources of unbounded memory consumption in functional programming languages: recursive data types, closures and recursive function calls. Although, all three are closely related when time and space program complexities are concerned, we discuss them separately because they constitute three important aspects of functional language design. The first category comprises types of data structures whose size is not fixed at compile time such as, for example, lists, trees or linked arrays. Typically, these data structures gain their size in purely functional programs as a result of their construction by recursive functions. Thus, their sizes are often related to the time complexity of these constructing functions. As a result, it is possible, using sophisticated type-theoretic techniques, to bound the amount of memory necessary for program to execute by constructing specific type deriviations for all program functions. For example, Hofmann & Jost (2003) propose a technique to automatically derive *linear* bounds on heap space usage of first-order functional programs while Vasconcelos & Hammond (2003) propose a technique addressing a class of recursive, higher-order functions whose cost (either space or time) can be expressed as a primitive-recursive function. Unfortunately, although of significant practical importance, these techniques cannot be used to analyse all function definitions that can possibly be expressed in the λ-calculus due to undecidability of the halting problem.

Closures are function definitions that are not closed, i.e., they reference variables outside of their definition scope, for example, internal variables of an enclosing function definition. When closures are used as values, as in many functional languages, these external variable references may escape their syntactic scope resulting in an additional memory cost because the garbage collector cannot free all of the memory of the enclosing function definition until it makes sure that all closures referencing it are no longer used. Since closures can be used to implement a wide range of recursive data structures (Abelson & Sussman 1996), the problem of bounding their memory cost is essentially that of programs with data structures of recursive types.

The use of recursive types and closures in a stream-based language that permits recursive stream definitions carries an additional risk. Because recursive streams are a form of infinite recursive function application, it is possible that unbounded size data structures be defined completely exhausting memory during program execution. Therefore, since investigation of the techniques for bounding memory use of such programs is outside of the scope of this work, we take preventive measures by disallowing both closures and recursive data types in the language. Due to the two-domain nature of SOSNA's type system, scalar function definitions cannot be nested and at the same time are not considered scalar values (e.g., cannot be returned by other functions as their results). Hence, closures cannot be defined and scalar functions can release all their memory once their evaluation finishes.

Recursive functions are in functional languages sources of uncertainty in both the time of execution and the memory used. As it was mentioned above, the undecidability of the halting problem makes it impossible, in the general case, to assert at compile time how long a given function would be executing and, as a result, how much memory it would need. Nevertheless, for many practical cases, it is possible to find an estimate. For example, Bonenfant et al. (2007) show how an accurate execution timing model of low-level abstract machine instructions, in conjunction with high-level source-based execution time bounds, can lead to accurate estimates of function execution times on concrete hardware architectures.

SOSNA takes a conservative approach and allows only those recursive calls that are made in the *tail* position, i.e., when there is no additional computation awaiting the result of the call. Tail-recursive functions have the well known property, that they can be transformed into looping statements that are computationally equivalent, but which have the convenient property of constant memory cost. This is caused by the fact that the looping statement can *update* the values of its temporary variables instead of having to allocate a new frame on the stack for each recursive call. Consider the following two implementations of a function computing the factorial of a natural number:

```
(def-rec (fact n)
  (if (> n 1)
    (* n (fact (- n 1)))
    1))
```

and

```
(def-rec (tail_fact n r)
  (if (> n 1)
    (tail_fact (- n 1) (* n r))
    r))
```

```
(def-fun (fact n)
  (tail_fact n 1))
```

In the first case, computing the factorial of a number n requires n partial results to be allocated on the stack because `fact` keeps recursively calling itself until `n` goes down to 1. In the other case, because the results are passed through function arguments the compiler can transform the function code into a loop expression executing in linear time but in constant memory.

SOSNA bounds execution time of its scalar functions by completely disallowing scalar function recursion when a special compilation parameter is set. This way SOSNA's scalar functions can be guaranteed to execute in bounded memory and time. It would be desirable, however, that a more sophisticated execution time estimation mechanism be implemented such as, for example, that of Bonenfant et al. (2007). Unfortunately, due to scope limitations, we have to leave this to the future work.

Stream Domain

As we briefly mentioned in the previous section, the SOSNA compiler in-lines all applications of stream functions removing them completely from the target program. This can be done when either recursive stream functions are disallowed or when the recursion is required to be bounded so that extensive in-lining could terminate. SOSNA takes the simpler approach and bans stream function recursion completely. As a result, programs are transformed to synchronous stream networks of static, known at compilation time size and structure that are guaranteed to execute in bounded memory (Caspi & Pouzet 1996).

Bounded Communication Extent

SOSNA implements all communication operators as sequences of a bounded and known at compile time number of neighbourhood data aggregation steps. Neighbourhood data aggregation rounds, on the other hand, are required to be executed within a bounded amount of time. As a result, all communication operators are executed in bounded time and the execution of each of the neighbourhood data aggregation rounds can be scheduled at compile time.

3.4 Programming in SOSNA

This section discusses a number of example programs in order to show how different types of WSAN applications could be implemented in SOSNA. Because SOSNA is only a prototype language targeting a relatively novel application domain it may not be suitable for development of WSAN applications all conceivable types. Therefore, the main intention of this section is to give some intuitions as to what can be achieved in the language rather than to try to map the whole WSAN application design space.

Amorphous Computing

Beal & Bachrach (2006) show how robust and biologically-inspired spatial programming can be achieved Proto, a stream-based programming language that uses neighbourhood stream aggregation as the only means of network communication. Spatial (amorphous) programs are constructed in Proto through the composition of *gradients* that are spatial flows of data values originating from a single node and radially expanding away throughout the network. The propagated data values typically increase with the distance from the source. The authors show that gradients are universal building blocks of spatial programs that are robust to node and communication failures, as well as, are adaptive to changes in the network topology. Proto implements gradients through recursive stream definitions involving neighbourhood stream aggregation.

Proto programs can be directly mapped to SOSNA because the language supports recursive stream definitions and neighbourhood stream aggregation. The following program defines a stream function implementing an unbounded-extent gradient through recursive-in-time neighbourhood stream aggregation.

```
(def (gradient src class)
  (letrec (n (+ 1 (IF src 0
                      (fby MAX_INT (aggr min MAX_INT (+ n 1) (nhood class))))))
     (- n 1)))
```

This program is a SOSNA translation of the original gradient implementation presented by Beal & Bachrach (2006). The function takes a Boolean stream (or a region in the spatial interpretation) and returns a stream/region that is equal to 0 at each node for which `(== src true)`. All nodes surrounding the source gradually increase their values as their hop distance from the source grows. The aggregation operator simply picks the minimum value of `(+ n 1)` from among those held by node's neighbours, starting from the `MAX_INT` constant that represents the maximum distance value the program can handle. The function is general, i.e., it can be used in any program because it is not tied to any particular device class definition.

Distributed Data Aggregation and Actuation

The two most fundamental tasks of any WSAN application are sensing and actuation. There are, however, many ways in which the flow of information can be organised in the network. For example, WSAN might comprise only autonomous units with sensing and actuation capabilities, or they might comprise sensor nodes whose readings are aggregated at one or more controllers that, in turn, communicate with actuators. Other applications might use only two types of devices - sensor nodes and controller/actuator nodes.

Consider the application presented in Listing 3.1 in which two different types of sensors - temperature sensors and presence detectors, together with a set of ventilators, the actuators, are deployed on two different floors of a building. The goal of this application is to minimise for each of the floors the use of ventilators by switching them on only when the temperatures exceed a threshold and when the presence detectors indicate there are people on the floor. We distinguish between floors to show how external constants, set during node programming can be used to differentiate between different nodes of the same type.

The program defines three device classes: a base station, a set of sensor nodes and ventilators. Each of the sensor nodes carries two sensors: a temperature sensor `temp` and a presence detector `presence`. The value of the external constant stream `floor` is assigned to each of the nodes when they are programmed, hence, we are able to differentiate among different devices of the same class. Similarly to constant streams, the external constant streams are initially assigned polymorphic types so that they can be referenced at devices of different classes.

The network operates as follows: in every instant, sensor nodes from each of the floors independently aggregate readings of their sensors by sending them towards the base station that computes the average temperatures and presence information for each of the floors separately. Knowing these values, the base station calculates a simple control law according to which ventilators at each of the floors are switched on when the average temperature exceeds 30 degrees while *at the same time* the presence detectors are sensing people presence. Note that the program makes no assumptions as to the exact number of sensors nodes and actuators present in the system, nor it makes any assumptions about the nature of the physical network topology. This way the network configuration can be augmented without the need of reprogramming of the already deployed devices. Nevertheless, if realistic control algorithms were to be used, knowledge of the number of nodes comprising different classes might become necessary. This, however, can easily be done in the same way as spatial stream averages are computed, i.e., through counting the nodes in parallel to other communication tasks.

The program demonstrates several things. Firstly, it implements a more sophisticated architecture

Listing 3.1: Distributed data aggregation and actuation SOSNA application.

```
(def-class base)
(def-class sensor_nodes : base)
(def-class vents : base)
(def-input temp @ sensor_nodes : int)
(def-input presence @ sensor_nodes : bool)

(def-extern floor : int)

(def (avg region sensor)
  (let (values (within region (vals sensor)))
    (let (sum (aggr + 0 values (clstat sensor_nodes)))
      (let (n (aggr + 0 (within region 1) (clstat sensor_nodes)))
        (/ sum n)))))

(def (present? region)
  (let (values (within region (vals presence)))
    (aggr or false values (clstat sensor_nodes))))

(def (ventilate region condition)
  (let (init (within region true))
    (expand and init condition (clstat vents))))

(def (needs-vent? region max-temp)
  (and (> (avg region) max-temp)
       (present? region)))

(def floor-1 (== floor 1))

(def floor-2 (== floor 2))

(output (Ventilator (merge (ventilate floor-1 (needs-vent? floor-1 30))
                           (ventilate floor-2 (needs-vent? floor-2 30)))))
```

in which the network comprises devices of three types that are grouped according to an application-specific criteria. Secondly, the program demonstrates how stream functions can be used to reuse repeatable stream transformation patterns and improve program readability. Finally, the program makes extensive use of stream filtering in order to control which network nodes should engage in collaborative stream processing tasks.

Target Tracking

Many macro-programming language publications use target tracking as a canonical application example. Target tracking is concerned with estimation of the position of a physical entity moving through an area. In the context of wireless sensor networks, target tracking involves computing a series of object's position estimates in a collaborative, local manner, i.e., by involving only those sensor nodes that are in object's proximity (Liu, Liu, Reich, Cheung & Zhao 2003). Some researchers consider also an extended scenario in which the tracking estimates produced by the network are used by a team of autonomous robots trying to catch the target (Schenato et al. 2005). We confine ourselves to the basic target tracking scenario and show how persistent state can be associated with the moving object so that its additional characteristics can be estimated. Listing 3.2 approximates object's position using the following centre of mass formula:

$$P_x = \frac{\sum_{i=0}^{N} x_i s_i}{\sum_{i=0}^{N} s_i} \qquad P_y = \frac{\sum_{i=0}^{N} y_i s_i}{\sum_{i=0}^{N} s_i},$$

where x_i, y_i are the coordinates of all nodes sensing the target and s_i are their respective sensor readings. We assume, that sensor nodes are assigned their physical positions offline and implement the persistent state as the number if instants in which the object is being detected in the network.

We assume that sensor readings above 100 imply object's presence in an area. The `obj_pos` function first establishes a dynamic cluster topology so that the node that is possibly the closest to the object can aggregate current sensor readings from the surrounding nodes literally implementing the centre of mass formula. Perhaps, the most important part of the program is the definition of the `distr_fby` function that implements a form of a distributed stream delay: the stream `s` is expected to be present only on `t`'s cluster heads and since different nodes might be taking the cluster head role, the function's task is to migrate the previous value of `s` to the current cluster head node. The state is not lost as long as previous cluster heads are within the current extent of the `t` topology. The function has a resetting behaviour, i.e., each time the object is lost (`s` takes a null value) or there is a communication failure, the initial value is used as a result so that the recursive stream `obj_age` can restart counting. Note, that because all definitions are dependent on the values of the

Listing 3.2: Target tracking SOSNA application.

```
(def-class nodes)
(def-input sensor @ nodes : int)
(def-extern pos_x : int)
(def-extern pos_y : int)
(def (obj_pos s t)
  (let (total_mass (aggr + s s t))
    (let (a (aggr + (* s pos_x) (* s pos_x) t))
      (let (b (aggr + (* s pos_y) (* s pos_y) t))
        (tuple (/ a total_mass) (/ b total_mass))))))
(def (distr_fby init s t)
  (let (s (fby init s))
    (let (forward (lambda (new old) new))
      (aggr forward (merge init s) s top))))
(def detection (within (> (vals sensor) 100)
                       (vals sensor)))
(def leader (clmax detection))
(def obj_x obj_y (obj_pos detection leader))
(def-rec obj_age (+ 1 (distr_fby 0 obj_age leader)))
(output (Report (tuple obj_x obj_y obj_age)))
```

`detection` stream, computations are performed by only those nodes that actually sense the object. Also, actuation is performed only by the cluster heads of the `leader` topology.

Distributed Feedback

Feedback is general concept in which past system output is used to influence the current or future system output. Feedback-based techniques are commonplace in control design, hence, it is reasonable consider feedback in the context WSANs. Listing 3.3 implements an adaptive threshold algorithm in which sensor nodes adjust their detection thresholds based on the the global average, computed by a base station. As a possible application, consider a case in which the background signal level, such as for example, sun light is changing throughout the day and event detection threshold must be adjusted accordingly. The following program uses the basic exponential moving average to smooth out subsequent estimates of the adaptive threshold according to the formula:

$$S_t = \alpha X_{t-1} + (1 - \alpha) S_{t-1}$$

The program calculates the average value of the `detection` stream similarly to the programs presented earlier in this section, i.e., by summing up all values and dividing the result by the number of received readings (the `avg` function). In order to explicitly handle the effects of possible communication failures the `avg` function rejects the computed result whenever it comprises data from less than six sensor nodes. Then, the `sustain` function is used (lines 22 and 24) to implement the substitution

Listing 3.3: Distributed feedback SOSNA application.

```
(def-class base)
(def-class nodes : base)
(def-input light @ nodes : float)

(def (avg s)
  (let (zero (tuple 0.0 0.0))
    (let ((tuple sum n)
            (aggr vector_add zero (tuple 1.0 s) (clstat nodes)))
      (within (> n 5)
        (/ sum n)))))

(def (exp_mavg s)
  (letrec (result (+ (* 0.2 s)
                     (* 0.8 (fby 0.0 result))))
    result))

(def (sustain init x)
  (letrec (y (merge (fby init y) x))
    y))

(def-rec detection
  (let (thresh_at_base (sustain 0.0 (fby 0.0 (exp_mavg (avg detection)))))
    (let (detection_thresh (expand max 0.0 thresh_at_base (clstat nodes)))
      (within (> (vals light) (sustain 0.0 detection_thresh))
        (vals ligth)))))

(output (Display (aggr max 0.0 detection (clstat nodes)))
```

of the previous stream value in the case the current one is absent. The function defines a recursive stream that is equal to the function's input stream each time it takes a non-null value, and that is equal to its previous value each time the input stream's value is absent.

The core part of this program is the definition of the `detection` stream that introduces two auxiliary streams. The `thresh_at_base` stream represents exponentially smoothed values of the global detection threshold and it is present only at the base station nodes, since its values involve the result of sensor data aggregation. The global detection threshold is defined over detection values from the *previous* instant (aggregation is performed under the **fby** operator) and its current values are propagated back to the sensor nodes (the `detection_thesh` stream) so that they can be used to filter out the *current* light sensor readings. As a result, the base station can display the maximum value of current detection reports.

Chapter 4

The SOSNA Compiler

The presentation of the SOSNA programming language given in the previous chapter remained somewhat abstract because many technical details related to the execution of SOSNA programs in WSANs were omitted. The main reason for such a treatment is that we want to decouple the language's semantics from its actual implementation so that it can be tailored to a particular target application execution platform. In this chapter, we present a prototype implementation of the SOSNA compiler, which currently supports two compilation targets: a custom-built SOSNA simulator and the NesC programming language. The simulator was written in the PLT Scheme programming language (Flatt & PLT 2009) and, in its current form, it allows to simulate SOSNA applications using a deterministic wireless communication model. While the simulator is intended for early application testing stages, the NesC target allows to run SOSNA programs on wireless networks of TinyOS-supported devices (so called *motes*) and serves as the primary means of evaluation of the synchronous macro-programming model.

The chapter is organised as follows. Section 4.1 describes the main program transformation stages and briefly discusses the limitations of the existing compiler. The global communication scheduling algorithm is presented in Section 4.1.10 and the rules governing duty cycling in SOSNA applications are described in Section 4.2.4. Section 4.1.15 describes a low-level abstract machine model and language that are used as the core of the NesC compilation target and which are intended both to amortise the memory cost of absent stream values, as well as, to serve as a portable platform for different C-like compilation targets. We conclude the chapter with a description of the SOSNA middleware which is a TinyOS library building on the Flooding Time Synchronization Protocol (Maróti et al. 2004) and driving the execution of SOSNA programs in mote networks.

4.1 Program Transformations

SOSNA is a high-level language, therefore, the gap between the semantics of the top-level program and its low-level realisation is large. The compiler implements a multi-stage transformation process that involves, among other tasks, translating the top-level program into the sequence of simpler languages, each of which being a step closer towards the compilation target language. The overall process is depicted in Figure 4.1 and it specifies both the transformation stages (oval boxes) and the most relevant intermediate languages (rectangles). The following subsections describe each element in more detail.

4.1.1 Parsing and Syntax Transformations

Because Sosna borrows its general syntax from Scheme, the parsing task can be significantly simplified. The compiler performs a two-phase process in which the top-level program is first transformed with a set of Scheme macros[1] in order to simplify its syntax and then it is parsed and translated to a simpler but equivalent language SOSNA Core. The macro transformations deal with translating all global program definitions into a nested sequence of **def** and **let** expressions. For example, the following program

```
(def-class base)
(def-input sensor @ base : int)
(def-fun (f x y)
  (+ x y))
(def result (smap f (vals sensor) 1))
(output (act result))
```

is translated to:

```
(def (class base)
  (def (input sensor base int)
    (let (f (lambda (x y) (+ x y)))
      (let (result (smap f (vals sensor) 1))
        (output (act result))))))
```

The nested sequence always ends with the **output** operator so that no program expressions can reference its result and, therefore, actuation can be scheduled at the end of the application schedule.

[1]High-level program syntax transformers. Since Scheme/Lisp macros are a well established programming concept, some people understand the term macro-programming as a programming style that makes extensive use of macros. This interpretation, however, bears little resemblance to the meaning ascribed to this term by the WSN/WSAN community.

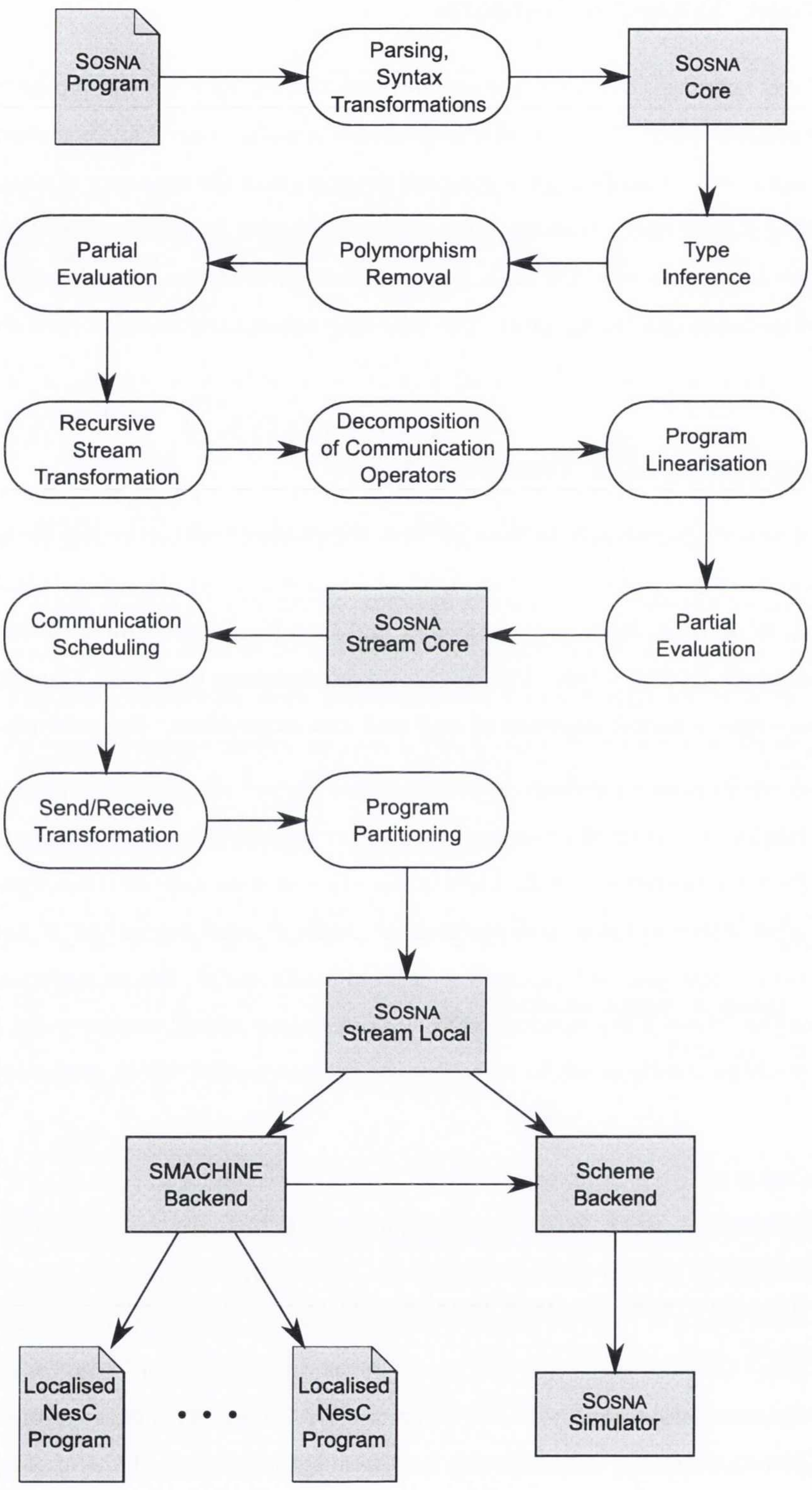


Figure 4.1: The main program transformation stages.

The main function of the parser is to modify the language's syntax with additional annotations for storing the type and source code information. The parser, however, performs two additional transformations. Firstly, judging by the syntactic context, the parser transforms all arithmetical stream expressions into the equivalent **smap** operator applications (see Section 3.2.1). Secondly, the parser removes all pattern matching expressions and all pattern variables replacing them with tuple element access operators. For example, the following function takes one argument that is a tuple of two elements whose values are assigned to the variables x and y.

```
(lambda((tuple x y))
  (+ x y))
```

After translation, the pattern is replaced with a new, internal variable and every reference to x and y is replaced with a tuple field access expression:

```
(lambda (%v)
  (+ (t2#1 %v) (t2#2 %v)))
```

The t2#1 and t2#2 operators return the first and the second element of a two-element tuple respectively and the NesC compilation target implements them as structure field access operators. For example, the following code would be generated for this function if we assume that both x and y are inferred to be of type Int.

```
int tmp_fun(struct _tuple_int_int _v){
    return _v.p1 + _v.p2;
}
```

4.1.2 SOSNA Core

The parser transforms the top-level SOSNA program into the simplified SOSNA Core language that serves as the input language for the type checker and which embodies the core SOSNA's concepts without the unnecessary *syntactic sugar*. SOSNA Core comprises two main classes of expressions that correspond to the language's scalar and stream domains and we denote them by e_f and e_s respectively. Figure 4.2 presents the grammar of the SOSNA Core language. For simplicity, we include only two-argument tuples and single-argument functions - the extension to the multi-argument case is straightforward.

The f expression class represents all language's built-in functions and it includes tuple constructors, tuple element accessors, arithmetical, logical and relational operators, as well as, numeric type conversions operators. The c expression class represents all language's constants (literals) while the

$$\begin{array}{lcl}
f & ::= & \texttt{tuple2} \mid \#^2_1 \mid \#^2_2 \mid + \mid - \mid \times \mid / \mid \ldots \\
c & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid 2 \mid \ldots \\
t & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \ldots \\
e_s & ::= & c \mid f \mid x \mid \lambda x.e_f \mid \Lambda x.e_s \mid e_s\ e_s \mid \texttt{rec}\ x.e_s \mid \texttt{let}\ x = e_s\ \texttt{in}\ e_s \mid \\
 & & \texttt{topclass}\ x \mid \texttt{subclass}\ x\ e_s \mid \texttt{extern}\ e_s\ t \mid \texttt{input}\ e_s\ t \mid \texttt{output}\ x\ e_s;\ \ldots\ x\ e_s; \mid \\
 & & \texttt{smap}\ e_s\ e_s \mid \texttt{fby}\ e_s\ e_s \mid \texttt{merge}\ e_s\ e_s \mid \texttt{within}\ e_s\ e_s \mid \texttt{at}\ e_s\ e_s \mid \texttt{vals}\ e_s \mid \texttt{time}\ e_s \mid \\
 & & \texttt{clstat}\ e_s \mid \texttt{clmin}\ e_s \mid \texttt{clmax}\ e_s \mid \texttt{nhood}\ e_s \mid \texttt{onehop}\ e_s\ e_s \mid \\
 & & \texttt{aggr}\ e_s\ e_s\ e_s\ e_s \mid \texttt{expand}\ e_s\ e_s\ e_s\ e_s \\
e_f & ::= & c \mid x \mid e_s\ e_f \mid \texttt{if}\ e_f\ \texttt{then}\ e_f\ \texttt{else}\ e_f \mid \texttt{let}\ x = e_f\ \texttt{in}\ e_f
\end{array}$$

Figure 4.2: The grammar of the SOSNA Core language.

t expression class represents all scalar type definitions used in specification of program inputs and external constants.

Stream domain expressions include respectively: constants (interpreted as constant streams), built-in functions, program identifiers, anonymous scalar functions, anonymous stream functions, stream function application expressions, recursive definitions, `let` definitions and references to all top-level program definitions, i.e., device classes, external constants, program inputs and outputs. The remaining expressions in the e_s class directly correspond to applications of all language stream and topology operators and their meaning was explained in Chapter 3.

The scalar domain is much more limited than the stream domain and it comprises respectively: constants (interpreted as scalar values), program identifiers, scalar function application and the conditional operator `if`. Note, that scalar functions belong to the stream domain and they can only be applied within the scalar domain. This property is reflected in the definition of the SOSNA's type system and is intended as a way of preventing the construction of streams of functions (see also Section 3.3.4).

4.1.3 Type Inference

In the previous chapter we introduced SOSNA's type system in terms of typing rules that defined types of the language's expressions in terms of the types of their sub-expressions. Such formulation, although precise, is not of much use when the sub-expression types are not known or are incomplete. Type inference (or type reconstruction) is a process that allows to discover the types of program expressions based on their syntax and semantics. Consider the following definition:

$$\texttt{let}\ x = (\texttt{within}\ y\ 10)\ \texttt{in}\ x$$

Knowing that the `within` operator takes two arguments, of which the first one must be a Boolean stream while the second can be a stream of an arbitrary data type, as well as, knowing that 10 represents a constant stream of integer numbers, we conclude that `x` must be a stream of integers. This reasoning describes the core idea behind type inference that can be a relatively simple procedure when programs comprise only expressions of simple types but which becomes more involved when more advanced concepts such as, for example, polymorphic types are introduced.

SOSNA follows the ML type discipline, hence its type system includes polymorphic types. One of the biggest advantages of using an ML-style type system is such that type inference is decidable and, although the worst-case inference complexity is exponential, efficient algorithms exist for the common case (Pierce 2004). The SOSNA compiler implements the constraint-based approach to ML type inference that is extensively presented in Chapter 10 of Pierce (2004).

Constraint-based typing is a technique that, instead of building a type derivation tree (a sequence of typing rule applications), constructs a set of type constraints, i.e., logical formulae defining certain relations among type variables. A program can be correctly typed only when its corresponding set of type constraints is satisfiable, i.e., there is an assignment of types to type variables such that all type constraints are satisfied. Constraint-based typing of ML-the-type-system is built around the concept of a constrained type scheme that is denoted as $\forall\bar{\mathsf{X}}[\mathcal{C}].\mathsf{T}$ and represents a generic type whose abstract type variables $\bar{\mathsf{X}}$ can be substituted only with the types that satisfy the constraint $\mathcal{C}$. The language of type constraints is defined as follows[2].

$$\begin{array}{lcl} \sigma & ::= & \forall\bar{\mathsf{X}}[\mathcal{C}].\mathsf{T} \\ \mathcal{C} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{T}_1 = \mathsf{T}_2 \mid \mathcal{C} \wedge \mathcal{C} \mid \exists\bar{\mathsf{X}}.\mathcal{C} \mid \mathsf{def}\ \mathsf{x} : \sigma\ \mathsf{in}\ \mathcal{C} \mid \mathsf{x} \preceq \mathsf{T} \end{array}$$

The meaning of the first four constraints is intuitive: true is an always satisfiable constraint, false is never satisfiable, $\mathsf{T}_1 = \mathsf{T}_2$ is satisfied only when T_1 is equal to T_2 and $\mathcal{C}_1 \wedge \mathcal{C}_2$ is satisfied when both $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied. The existential constraint $\exists\bar{\mathsf{X}}.\mathcal{C}$ states that there must exist a substitution of types to type variables $[\vec{\mathsf{X}} \mapsto \vec{\mathsf{T}}]$ such that the constraint $\mathcal{C}$ can be satisfied. Also, all type variables in $\bar{\mathsf{X}}$ are considered to be bound in the definition of $\mathcal{C}$. The next constraint simply binds a type scheme to the identifier x in the definition of the constraint $\mathcal{C}$ and it is satisfied only when $\mathcal{C}$ is satisfied under the given binding of x. The last constraint defines type scheme substitution (instantiation) and it is satisfied only when the type T is an *instance* of the type scheme bound to x. A type T is an instance of a type scheme $\forall\bar{\mathsf{X}}[\mathcal{C}].\mathsf{T}'$ if and only if $\exists\bar{\mathsf{X}}.(\mathcal{C} \wedge \mathsf{T} = \mathsf{T}')$ is satisfiable. Additionally, we use the two

[2]We simplify the abstract presentation given in Pierce (2004) in order to better reflect the implementation.

following abbreviations:

$$\begin{aligned} \forall \bar{\mathsf{X}}.\mathsf{T} &\equiv \forall \bar{\mathsf{X}}[\mathsf{true}].\mathsf{T} \\ \mathsf{let}\ \mathsf{x} : \forall \bar{\mathsf{X}}[\mathcal{C}].\mathsf{T}\ \mathsf{in}\ \mathcal{C}' &\equiv \exists \bar{\mathsf{X}}.\mathcal{C} \wedge (\mathsf{def}\ \mathsf{x} : \forall \bar{\mathsf{X}}[\mathcal{C}].\mathsf{T}\ \mathsf{in}\ \mathcal{C}'). \end{aligned}$$

The type inference algorithm constructs a single type constraint $\mathcal{C}$ based on the structure of the input program and then it uses a unification-based constraint solving algorithm to find a satisfiable assignment of types to all program type variables referenced in $\mathcal{C}$. We use the function $[\![\cdot]\!]$ to represent the constraint generation process and for the case of ML-the-calculus we define it as follows.

$$\begin{aligned} [\![x : \mathsf{T}]\!] &= x \preceq \mathsf{T} \\ [\![\lambda x.e : \mathsf{T}]\!] &= \exists \mathsf{X}_1\mathsf{X}_2.(\mathsf{let}\ x : \mathsf{X}_1\ \mathsf{in}\ [\![e : \mathsf{X}_2]\!] \wedge \mathsf{X}_1 \rightarrow \mathsf{X}_2 = \mathsf{T}) \\ [\![e_1\ e_2 : \mathsf{T}]\!] &= \exists \mathsf{X}.([\![e_1 : \mathsf{X} \rightarrow \mathsf{T}]\!] \wedge [\![e_2 : \mathsf{X}]\!]) \\ [\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!] &= \mathsf{let}\ x : \forall \mathsf{X}[[\![e_1 : \mathsf{X}]\!]].\mathsf{X}\ \mathsf{in}\ [\![e_2 : \mathsf{T}]\!] \end{aligned}$$

The $[\![\cdot]\!]$ relation builds the type constraint for the whole program by recursively traversing its structure. Because Sosna assigns different types to certain expressions based on their syntactic context (see Section 3.3), we use two different constraint generation procedures to type the constructs of the scalar and stream domains respectively. For example, language literals such as integer constants are typed as constant scalar values within function bodies and as constant streams elsewhere. The following two constraint generation rules explain the idea.

$$\begin{aligned} [\![\mathtt{1} : \mathsf{T}]\!]_f &= (\mathsf{T} = \mathsf{Int}) \\ [\![\mathtt{1} : \mathsf{S}]\!]_s &= \exists \mathsf{X}_\mathsf{C}.(\mathsf{S} = \mathsf{Stream}(\mathsf{Int}, \mathsf{X}_\mathsf{C})) \end{aligned}$$

The second rule shows also how polymorphic types of constant streams are handled. As we explained in Section 3.3, constant streams are assigned the following types schemes $\sigma = \forall \mathsf{X}_\mathsf{C}.\mathsf{Stream}(\mathsf{T}, \mathsf{X}_\mathsf{C})$. Each use of a constant stream in a program defines this stream's instance therefore, in accordance with the semantics of type scheme substitution we can say that

$$\sigma \preceq \mathsf{S} \equiv \exists \mathsf{X}_\mathsf{C}.\Big(\mathsf{true} \wedge \mathsf{S} = \mathsf{Stream}(\mathsf{T}, \mathsf{X}_\mathsf{C})\Big) \equiv \exists \mathsf{X}_\mathsf{C}.(\mathsf{S} = \mathsf{Stream}(\mathsf{T}, \mathsf{X}_\mathsf{C}))$$

Additionally, we discuss the typing of an example expression in order to give some intuitions of how the inference process works. Consider the following Sosna Core expression:

$$\mathtt{let}\ (f\ \lambda x.1)\ (\mathtt{smap}\ f\ 1.0)$$

The expression defines a function f that takes one argument of an unspecified type and that returns the value 1 (of type Int) as a result. The function is applied point-wise in time to a constant stream of floating point numbers so that the overall result of this expression is a constant stream of integer values. The following rule defines type constraint generation for `smap` operator (single-argument case).

$$[\![\texttt{smap}\ e_1\ e_2 : \mathsf{S}]\!]_s \quad = \quad \exists \mathsf{X_T X'_T X_C}. \begin{pmatrix} [\![e_1 : \mathsf{X_T} \rightarrow \mathsf{X'_T}]\!]_s \wedge \\ [\![e_2 : \mathsf{Stream}(\mathsf{X_T}, \mathsf{X_C})]\!]_s \wedge \\ \mathsf{S} = \mathsf{Stream}(\mathsf{X'_T}, \mathsf{X_C}) \end{pmatrix}$$

The constraint generation process starts with the top-level expression:

$$\begin{aligned} [\![\texttt{let}\ f = \lambda x.1\ \texttt{in}\ (\texttt{smap}\ f\ 1.0) : \mathsf{S}]\!]_s \quad &= \quad \mathsf{let}\ f : \forall \mathsf{X_S}[[\![\lambda x.1 : \mathsf{X_S}]\!]_s].\mathsf{X_S}\ \mathsf{in} \\ & \qquad [\![(\texttt{smap}\ f\ 1.0) : \mathsf{S}]\!]_s \end{aligned}$$

then type scheme constraints are generated:

$$\begin{aligned} &\mathsf{let}\ f : \forall \mathsf{X_S}[\exists \mathsf{X_T X'_T}.(\mathsf{let}\ x : \mathsf{X_T}\ \mathsf{in}\ [\![1 : \mathsf{X'_T}]\!]_\mathsf{f} \wedge \mathsf{X_T} \rightarrow \mathsf{X'_T} = \mathsf{X_S})].\mathsf{X_S}\ \mathsf{in} \\ &\quad [\![(\texttt{smap}\ f\ 1.0) : \mathsf{S}]\!]_s \end{aligned}$$

and simplified:

$$\mathsf{let}\ f : \forall \mathsf{X_S X_T}[\mathsf{X_S} = \mathsf{X_T} \rightarrow \mathsf{Int})].\mathsf{X_S}\ \mathsf{in}\ [\![(\texttt{smap}\ f\ 1.0) : \mathsf{S}]\!]_s$$

Finally, the bodies of the let constraints are processed and simplified:

$$\begin{aligned} &\mathsf{let}\ f : \forall \mathsf{X_S X_T}[\mathsf{X_S} = \mathsf{X_T} \rightarrow \mathsf{Int})].\mathsf{X_S}\ \mathsf{in} \\ &\quad \exists \mathsf{X_C}.([\![f \preceq \mathsf{Float} \rightarrow \mathsf{Int}]\!]_s \wedge \mathsf{S} = \mathsf{Stream}(\mathsf{Float}, \mathsf{X_C})) \end{aligned}$$

As a result, the function f received a polymorphic type that allows values of any type to be passed as arguments. The function is referenced only once hence it has only one instance that receives a concrete type $\mathsf{Float} \rightarrow \mathsf{Int}$. Note that the overall expression type is incomplete since it does not specify the device class on which the `smap` operator is evaluated. Such cases can be resolved in two ways: either the value of the $\mathsf{X_C}$ variable is discovered later (if this expression is a part of a larger one which provides the missing information) or the compiler reports an error suggesting the use of the `at` operator in order to explicitly name the device class.

After the program is typed, the compiler annotates every expression with its type but for clarity of presentation we omit these annotations in the following sections whenever this is possible. The knowledge of program types allows the compiler to control the use of recursion: programs that declare recursive stream functions can be rejected, recursive scalar function calls can be enforced to be in tail position (see Section 3.3.4) and recursive stream references can be accepted only when they occur in the context of the `fby` operator. Also, at this stage, the compiler can see whether all program expressions have complete types and reject those that do not. As a result, the `at` operator becomes redundant and is removed from the program.

The output of the type inference module conforms to the SOSNA Core language. However, subsequent program transformations, as described in the following sections, introduce program constructs that are not part of the SOSNA Core language but rather correspond to the intermediate steps of the translation from SOSNA Core to the SOSNA Stream Core language which is summarised in Section 4.1.13.

4.1.4 Polymorphism Removal

SOSNA assigns polymorphic types to all `let`-bound identifiers so that it is possible to use them in different contexts, for example, functions can have partially-defined signatures and constant streams can be used in expressions from different device classes. This property, however, although useful, poses certain challenges when the implementation of such polymorphic constructs is considered. The following example SOSNA Core program defines a recursive stream whose type is incomplete with respect to its device class. The stream is used in two different contexts hence two type instances are created, each one comprising a different device class subtype.

$$\texttt{let}\ n = \texttt{rec}\ n.(\texttt{smap}\ +\ 1\ (\texttt{fby}\ 0\ n))\ \texttt{in}$$
$$\texttt{output display at}\ c_0\ n;\ \texttt{blink at}\ c_1\ n;$$

The language's semantics disallow the same stream to be evaluated on devices of two different classes. Therefore, the only reasonable way of handling this problem is to assume that there are two different instances of `n`, each one evaluated on devices of a different class. This is exactly how the compiler handles `let` polymorphism: for every `let`-bound identifier, all of its instances are collected and copies of this identifier's expression are created and inserted to the program. As a result of this process, all `let` expressions are replaced with sequences of `def` expressions that have the same function but which require their identifiers to have concrete types. The above program would be translated as follows.

$$\texttt{def}\ n_1 = \texttt{rec}\ n_1.(\texttt{smap}\ +\ 1\ (\texttt{fby}\ 0\ n_1))\ \texttt{in}$$
$$\texttt{def}\ n_2 = \texttt{rec}\ n_2.(\texttt{smap}\ +\ 1\ (\texttt{fby}\ 0\ n_2))\ \texttt{in}$$
$$\texttt{output display}\ n_1;\ \texttt{blink}\ n_2;$$

The duplication process involves only identifier instances which receive different types. For example, if in the above example, the stream n was referenced twice but was assigned the same device class in both cases, only one definition would be added. Additionally, the algorithm, as a by-product, removes all `let` bindings that are not referenced in their corresponding `let` body expressions.

4.1.5 Partial Evaluation

After program types have been concertised, the compiler removes all stream functions through extensive in-lining of their definitions. The process is guaranteed to terminate because at this stage none of the program's stream functions are recursive. Banning stream function recursion is a design choice that does not have much justification beyond the author's preference. An alternative approach would be that of the Regiment macro-programming system (Newton et al. 2007) that requires recursion to be bounded, i.e., to terminate after a fixed number of steps so that the in-lining process can terminate. We do not see this aspect as essential to the overall contribution of the work, therefore, we chose the first approach. However, there is no technical objection that bounded recursion in-lining could be implemented within the existing compiler.

The in-lining procedure uses `def` expressions to limit the size of the resulting program. Consider the following example program that defines a stream function of one argument.

$$\texttt{let}\ f = \Lambda x.(\texttt{smap} + x\ x)\ \texttt{in}$$
$$f\ (\texttt{smap} \times 2\ 2)$$

In-lining of this function results in the following expression being generated:

$$\texttt{def}\ x = (\texttt{smap} \times 2\ 2)\ \texttt{in}$$
$$\texttt{smap} + x\ x$$

We can see that this way the expression `smap` $\times$ 2 2 is evaluated only once, before the evaluation of the function's body starts, exactly as it would be the case during function evaluation at runtime.[3]

Along stream function in-lining, the partial evaluation procedure evaluates all constant expressions and removes all constant `def` bindings. Therefore, the above example, after the function has been in-lined, reduces to the constant stream of the value 8.

4.1.6 Recursive Stream Transformation and Initialisation

Stream recursion in synchronous languages does not lead to unbounded memory consumption but rather can be efficiently implemented with little memory overhead. Recursive streams are defined in SOSNA with the help of the `fby` operator whose task is to delay all references to the recursive stream identifier within the context of its definition. The operator can be thought of as defining a one-element buffer that stores the current value if its second argument (the delayed stream). In every instant, the operator uses the contents of the buffer as its current result while saving the current

[3]Some functional languages delay the evaluation of function arguments until they are actually needed and this approach is called *lazy evaluation*. SOSNA takes a more conservative approach and uses *strict evaluation* order in which arguments are evaluated from the first to the last.

value of its argument for use in the subsequent instant. The current version of the SOSNA compiler does not apply any particular optimisations and implements the `fby` operator in a simple way, i.e., by assigning a permanent state variable to every occurrence of the operator for use as its buffer. We use the following expression to explain how recursive stream definitions and the `fby` operator are implemented.

$$\texttt{rec}\ n.(\texttt{smap} + 1\ (\texttt{fby}\ 0\ n)$$

The translation first generates an initialisation expression for the `fby` buffer, then the value of the recursive stream is computed and *after* that the buffer is updated with the newly computed value of the delayed expression. When this program is run repetitively in a loop (recall from Section 3.1 how synchronous programs are executed), then this procedure essentially realises the recursive semantics.

```
init $fby_0 = 0$ in
  def $n$ = smap + 1 $fby_0$ in
    set_after $n$ $fby_0 = n$ in
      $n$
```

The transformation procedure moves all `init` expressions to the top-level so that they can be implemented as a global program initialisation procedure. To this end, the initialisation expressions must necessarily be constant. The compiler checks, therefore, whether all delayed streams are initialised with constant values and, as an optimisation, it evaluates all arithmetical expressions that are defined over constant streams.

The `set_after` expression changes the value of the given `fby` buffer requiring at the same time that this operation must be performed after the evaluation of its first argument has finished. Although this is clearly the case in the above program, the requirement gains significance during the operation of the communication scheduling algorithm that relies on data dependencies between different expressions.

4.1.7 Decomposition of Communication Operators

Section 3.2.2 explained the semantics of the stream aggregation and propagation operators in terms of sequences of single-hop data aggregation rounds. We also said that in the case of homogeneous neighbourhood topologies stream aggregation is equivalent to stream propagation. The SOSNA compiler translates all `aggr` and `expand` operators to sequences of applications of a universal, neighbourhood data aggregation operator `staggr`. The operator's semantics are general so that homogeneous and heterogeneous stream data aggregation and propagation can be accommodated. The core idea behind this unification is the low-level representation of topologies as tuples comprising a local node's parent's network identifier and the level the node occupies in the topology. This representation is natural for cluster or tree topologies because the parent's identifier and the distance to the tree's root node are the least every topology member needs to know. For neighbourhood topologies, however, we use broadcast addresses as parent identifiers because a single parent node can no longer be distinguished. We use the term *topology descriptor* to refer to this low-level topology representation.

Similarly to the `aggr` and `expand` operators, `staggr` uses the presence and absence patterns of the initialisation and the aggregated streams to decide at runtime which nodes should participate in the operator's evaluation (see Section 3.2.2 for details). Therefore, the main function of the operator is to translate high-level topology expressions into their low-level representations and to serve as a universal building block of all SOSNA's communication primitives. In this sense, `staggr` implements the abstract and universal neighbourhood aggregation operation, as defined by the synchronous macroprogramming model in Section 3.1. The operator has the following syntax:

$$\texttt{staggr}\ f\ i\ s\ r\ d\ l_R\ l_S$$

The first three arguments are the aggregation function, the initialisation and the aggregated streams representing the first three arguments of the corresponding `aggr` or `expand` operators and their semantics is exactly the same as that presented in Section 3.2.2. The remaining arguments constitute the topology's low-level representation. Their semantics is the following: $r = (r_p, r_l)$ represents the receiver's topology descriptor and $d = (d_p, d_l)$ represents the destination topology descriptor. Both are pairs of values of which the first is a network address while the second represents the level a node occupies in the topology. The network address has a different meaning is each case: it is the node's parent in the case of the destination descriptor and it is the node's own network identifier in the case of the receiver's descriptor. The last two arguments of the operator, specify the receiving and sending topology levels, i.e., they define which topology levels should be receiving and which should be sending data during the operator's evaluation. This information is sufficient for the sending nodes to know when to send and whom to send to, as well as, for the receiving nodes, to know when to receive and whom to receive from. The operator's arguments have the following types (for certain T, T', X_C and X'_C):

$$f : (\mathsf{T}', \mathsf{T}) \rightarrow \mathsf{T} \qquad d : \mathsf{Stream}(\mathsf{NetId} \times \mathsf{Byte}, \mathsf{X}'_\mathsf{C})$$
$$i : \mathsf{Stream}(\mathsf{T}, \mathsf{X}_\mathsf{C}) \qquad l_R : \mathsf{Stream}(\mathsf{Byte}, \mathsf{X}_\mathsf{C})$$
$$s : \mathsf{Stream}(\mathsf{T}', \mathsf{X}'_\mathsf{C}) \qquad l_S : \mathsf{Stream}(\mathsf{Byte}, \mathsf{X}'_\mathsf{C})$$
$$r : \mathsf{Stream}(\mathsf{NetId} \times \mathsf{Byte}, \mathsf{X}_\mathsf{C})$$

In Section 3.2.2 we defined the operational semantics of neighbourhood stream aggregation and propagation in terms of the presence/absence patterns of the initialisation and the aggregated streams. We implicitly assumed that the definition applied to all nodes that were members of the given neighbourhood topology. The semantics of the `staggr` operator make no additional assumptions because the operator uses a concrete topology representation instead of an abstract one. In order to give the operator's semantics we introduce the following auxiliary functions:

- $\mathit{current}(s)$ returns the current value of the stream s or the null value if s is absent in the current instant,
- $\mathit{present}(s)$ returns `true` when the current value of the stream s is present, and `false` otherwise,
- $\mathit{devclass}(s)$ returns the stream's device class identifier (e.g., an integer value) and $\mathit{devclass}(\texttt{self})$ represents the node's own device class.

The operational semantics of a single round of neighbourhood stream aggregation denoted as:

$$\texttt{staggr}\ f\ i\ s\ (r_p, r_l)\ (d_p, d_l)\ l_R\ l_S$$

are the following. In every instant:

1. A node sends the tuple $(\mathit{current}(s), d_p)$ when $\mathit{devclass}(\texttt{self}) = \mathit{devclass}(s)$ and when:

$$\mathit{present}(s) \wedge \mathit{present}((d_p, d_l)) \wedge d_l = l_S$$

2. A node receives the tuple (v, p) when $\mathit{devclass}(\texttt{self}) = \mathit{devclass}(i)$ and when:

$$\mathit{present}(i) \wedge \mathit{present}((r_p, r_l)) \wedge r_p = p \wedge r_l = l_R$$

Neighbourhood stream aggregation and propagation are the simplest to decompose. For example, the following expression

$$\texttt{aggr} + 0\ 1\ (\texttt{nhood}\ m)$$

is translated to

$$\texttt{def}\ m' = \texttt{at}\ m\ (\texttt{smap tuple2 bcast 0})\ \texttt{in}$$
$$\texttt{staggr} + 0\ 1\ m'\ m'\ 0\ 0$$

where m' is a topology descriptor corresponding to all neighbours from the m device class. We use the `at` operator only for presentation purposes in order to show that the topology descriptor is evaluated within the m device class. At this stage of program transformation, all instances of the `at` operator have been removed and the newly introduced constructs are explicitly assigned their types to match the typing of other program expressions. The values of the receiver's and the destination topology descriptors, as well, as the values of the sending and receiving levels indicate that all nodes from the m device class should engage in communications and the result is determined by the presence/absence patterns of the initialisation and the aggregated streams, in exactly the same way as it was presented in Section 3.2.2.

Heterogeneous neighbourhood stream aggregation is handled similarly, but with the difference that the receiver's and the destination topology descriptors belong to different device classes. We vary the topology levels in order to break the symmetry between both device classes so that nodes in the sending device class constitute the level 1 in the topology and the receiving nodes constitute the level 0. For example, the following expression

$$\texttt{aggr} + 0\ 1\ (\texttt{onehop}\ m_R\ m_S)$$

is translated to

$$\begin{array}{l} \texttt{def}\ m'_R = \texttt{at}\ m_R\ (\texttt{smap tuple2 bcast 0})\ \texttt{in} \\ \quad \texttt{def}\ m'_S = \texttt{at}\ m_S\ (\texttt{smap tuple2 bcast 1})\ \texttt{in} \\ \quad\quad \texttt{staggr} + 0\ 1\ m'_R\ m'_S\ 0\ 1 \end{array}$$

Decomposition of multi-hop stream aggregation and propagation operators follows closely the process explained in Section 3.2.2: different tree levels are activated at different stages of the process and values are gradually flowing towards cluster heads, in the case of stream aggregation, and outwards from cluster heads, in the case of stream propagation. We use heterogeneous data aggregation to exemplify the decomposition process and in the following example we assume that the static cluster topology has a maximum depth of three hops, we use the $superclass(m)$ function to denote the super class of the device class m, and we omit for clarity the second argument of the `extern` operator.

$$\texttt{aggr} + i\ s\ (\texttt{clstat}\ m)$$

is translated to

$$\begin{array}{l} \texttt{def}\ m_S = \texttt{at}\ m\ (\texttt{extern}\ static\_cluster)\ \texttt{in} \\ \quad \texttt{def}\ m_R = \texttt{smap tuple2}\ (\texttt{extern}\ id)\ (\#_2^2\ m_S)\ \texttt{in} \\ \quad\quad \texttt{def}\ m'_R = \texttt{at}\ superclass(m)\ (\texttt{smap tuple2}\ (\texttt{extern}\ id)\ 0)\ \texttt{in} \\ \quad\quad\quad \texttt{def}\ r_0 = \texttt{staggr} + s\ s\ m_R\ m_S\ 2\ 3\ \texttt{in} \\ \quad\quad\quad\quad \texttt{def}\ r_1 = \texttt{staggr} + s\ r_0\ m_R\ m_S\ 1\ 2\ \texttt{in} \\ \quad\quad\quad\quad\quad \texttt{staggr} + i\ r_1\ m'_R\ m_S\ 0\ 1\ \texttt{in} \end{array}$$

The *static_cluster* external definition represents the static topology's descriptor, as reported by the underlying middleware. This value is not constant and may change in time along with the wireless network connectivity. Heterogeneous data aggregation and propagation use two different receiver's topology descriptors: m_R is evaluated on cluster members that, depending on the level they occupy in the topology, take also the receiving role, m'_R is evaluated on the topology's cluster heads which, since they belong to the different device class - *superclass*(m), have to be handled differently. The former is defined as a tuple comprising the node's own network identifier and the level it currently occupies within the static topology. This way the receiving nodes will receive data values only from their tree children, i.e., nodes that are one level below them and which define them as the destinations for their data items. The latter is defined in the same way but the definition does not reference *static_cluster* since, on a different device class, it refers to a different cluster topology.

The above process closely matches the semantics of multi-hop stream aggregation given in Section 3.2.2. Heterogeneous stream propagation is defined analogously but with the difference that the sending and receiving roles are reversed. Homogeneous stream aggregation and propagation are translated in a similar way but with the main difference that the *static_cluster* external definition is replaced with the dynamic cluster topology descriptor that is computed as a result of the evaluation of the `clmax` and `clmin` operators. Also, since all topology descriptors are evaluated within the same device class, there is no need for two different receiver's topology descriptors. The following example explains the idea.

$$\texttt{aggr} + i\ s\ (\texttt{clmax}\ (\texttt{vals sensor}))$$

is translated to

```
def $m_S$ = clmax (vals sensor)) in
  def $m_R$ = smap tuple2 (extern $id$) ($\#^2_2$ $m_S$) in
    def $r_0$ = staggr + $s$ $s$ $m_R$ $m_S$ 2 3 in
      def $r_1$ = staggr + $s$ $r_0$ $m_R$ $m_S$ 1 2 in
        staggr + $i$ $r_1$ $m_R$ $m_S$ 0 1 in
```

Dynamic cluster topologies are constructed through the leader election process that involves a constant number of neighbourhood stream aggregation rounds in which both the leader and the topology are established. The process comprises a sequence of n `staggr` expressions that aggregate tuples of three elements: the current value of the selection stream, the current parent node's network identifier and the current distance to the topology's root node. The following code presents the translation of the (`clmax` s) expression.

```
def $m'$ = within $present(s)$ (smap tuple2 bcast 0) in
  def $r_0$ = smap tuple3 $s$ (extern $id$) 0 in
    def $r_1$ = staggr $S_{max}$ $r_0$ $r_0$ $m'$ $m'$ 0 0 in
      def $r_2$ = staggr $S_{max}$ $r_1$ $r_1'$ $m'$ $m'$ 0 0 in
        def $r_3$ = staggr $S_{max}$ $r_2$ $r_2'$ $m'$ $m'$ 0 0 in
          smap tuple2 (smap $\#_2^3$ $r_3$) (smap $\#_3^3$ $r_3$)
```

The leader election process is performed along the neighbourhood topology that is trimmed to the group of nodes for which the selection stream takes a non-null value (hence the `within` operator). Initially, all participating nodes consider themselves to be the topology's roots (r_0). Then, in each aggregation round, this view is updated when nodes receive tuples with better selection stream's values (r_1, r_2, r_3). The updated results are resent in the next round with a small modification that puts the node's network identifier as the third element of the tuple so that the the node's neighbours can store it as their parent's identifier:

```
$r1'$ = smap tuple3 ($\#_1^3$ $r_1$) ($\#_2^3$ $r_1$) (extern $id$)
$r2'$ = smap tuple3 ($\#_1^3$ $r_2$) ($\#_2^3$ $r_2$) (extern $id$)
```

Eventually, the node with locally the best value becomes the topology's root because it did not receive a better value than its own while all other nodes that are within the distance of three hops from the root node consider themselves topology members and know their distance to the root, as well as, the network identifier of their tree parent. The final result, i.e., the dynamic topology descriptor, is computed by extracting the second and the third component from the result of the last aggregation round (r_3).

The selection logic is implemented by either the S_{max} or the S_{min} functions and their definitions are added to the program when required. The following pseudo-code defines the S_{max} function (the S_{min} function replaces the first $>$ relation with $<$).

```
$S_{max}$ $(s, id, d)$ $(s_{max}, id_{max}, d_{max})$ =
  if $s > s_{max}$ then
    return $(s, id, d + 1)$
  else
    if $(s = s_{max}) \wedge (d + 1 < d_{max})$ then
      return $(s, id, d + 1)$
    else
      return $(s_{max}, id_{max}, d_{max})$
```

The code is mainly self-explanatory. We only note that the leader election process, in addition to selecting the node with the locally-best value of the selection stream, minimises the the distance to the root in situations where there are more than one leader with the same value of the selection stream within the n hops from each other.

4.1.8 Program Linearisation

Program transformations presented in the previous sections may add many auxiliary stream definitions. For the purpose of scheduling the communication operators, the compiler needs to simplify the program's structure and it does so by restructuring nested `def` expressions so that they can be considered sequential. Consider the following example expression:

$$\begin{array}{l}\texttt{def } x = \texttt{def } y_1 = e_1 \texttt{ in} \\ \qquad\qquad\quad \texttt{def } y_2 = e_2 \texttt{ in } e_3 \\ \texttt{in } e_4\end{array}$$

Such forms are legal in both SOSNA and SOSNA Core and can be introduced by both the programmer or as a result some of the program transformation stages, e.g., the decomposition of communication operators. In order to restructure the program in the way that reflects the order in which expressions are evaluated, the compiler performs the following basic transformation:

$$\begin{array}{l}\texttt{def } y_1 = e_1 \texttt{ in} \\ \quad \texttt{def } y_2 = e_2 \texttt{ in} \\ \qquad \texttt{def } x = e_3 \texttt{ in } e_4\end{array}$$

Additionally, the compiler ensures that all `staggr` operator instances have the following general form:

$$\texttt{def } x = \texttt{staggr } \ldots \texttt{ in } e$$

as well as, it assigns temporary names to all anonymous functions and moves all function definitions to the top of the `def` sequence, preserving their mutual dependencies. As a result, the whole program is structured as a nested sequence of `init`, `def` and `set_after`, expressions and each of the identifier expressions are simple, i.e., they do not contain any of the above three nesting forms. The most-nested expression, i.e., the one that is evaluated at the end of the evaluation loop is the `output` expression.

4.1.9 SOSNA Stream Core

Linearisation is the last of the sequence of program transformations that gradually translate a SOSNA Core program to the SOSNA Stream Core language. SOSNA Stream Core is a statically typed formalism

$$
\begin{array}{lcl}
f & ::= & \texttt{tuple2} \mid \#^2_1 \mid \#^2_2 \mid + \mid - \mid \times \mid / \mid \ldots \\
c & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid 2 \mid \ldots \\
e_i & ::= & \texttt{init}\ x = c\ \texttt{in}\ (e_i \mid e_d) \\
e_d & ::= & \texttt{def}\ x = e_s\ \texttt{in}\ e_d \mid \texttt{set\_after}\ x\ x = e_s\ \texttt{in}\ e_d \mid \texttt{output}\ x\ e_s;\ \ldots\ x\ e_s; \\
e_s & ::= & c \mid f \mid x \mid \lambda x.e_f \mid \texttt{rec}\ x.\lambda x.e_f \mid \texttt{extern}\ x \mid \texttt{input}\ x \mid \\
 & & \texttt{smap}\ e_s\ e_s \mid \texttt{merge}\ e_s\ e_s \mid \texttt{within}\ e_s\ e_s \mid \texttt{vals}\ e_s \mid \texttt{time}\ e_s \mid \\
 & & \texttt{staggr}\ e_s\ e_s\ e_s\ e_s\ e_s\ e_s\ e_s \\
e_f & ::= & c \mid x \mid e_s\ e_f \mid \texttt{if}\ e_f\ \texttt{then}\ e_f\ \texttt{else}\ e_f \mid \texttt{def}\ x = e_f\ \texttt{in}\ e_f
\end{array}
$$

Figure 4.3: The grammar of the SOSNA Stream Core language.

in which programs can be seen essentially as sequences of stream expressions that are simple enough to be fully evaluated within the duration of a single *round* of neighbourhood data aggregation. This property is used by the scheduling algorithm to assign different expressions for evaluation during different data aggregation rounds resulting in a global communication and computation schedule.

Figure 4.3 presents the language's grammar. Compared to SOSNA Core (Figure 4.2) the grammar introduces two additional expression classes e_i and e_d that reflect the linear structure of SOSNA Stream Core programs, as well as, they reflect the fact that the initialisation expressions are the top-most program expressions. The stream expression category e_s has been significantly simplified so that scheduling can be performed while the scalar domain was left unchanged because SOSNA considers the scalar domain independent of the stream domain so that in the future versions of the language the scalar domain can extend it.

4.1.10 Communication Scheduling

The current version of the SOSNA compiler implements a communication scheduling policy that tries to minimise the overall schedule length by scheduling several neighbourhood stream aggregation expressions for execution in parallel. The algorithm analyses program data dependencies trying to maximise the degree of parallelism shortening this way the overall application cycle's length.

Data dependency is a relationship between two program identifiers in which one references another. Because recursive stream definitions have been already decomposed, SOSNA Stream Core programs cannot have recursive data dependencies. Therefore, data dependency analysis, when applied to the whole program, results in computation of the data dependency graph that is an acyclic, directed graph representing data dependencies among different program identifiers. The scheduling algorithm applies the data dependency analysis at the granularity of `init`, `def`, `set_after` and `output` identifiers treating their corresponding expressions as a basic unit of granularity. As an example, consider the

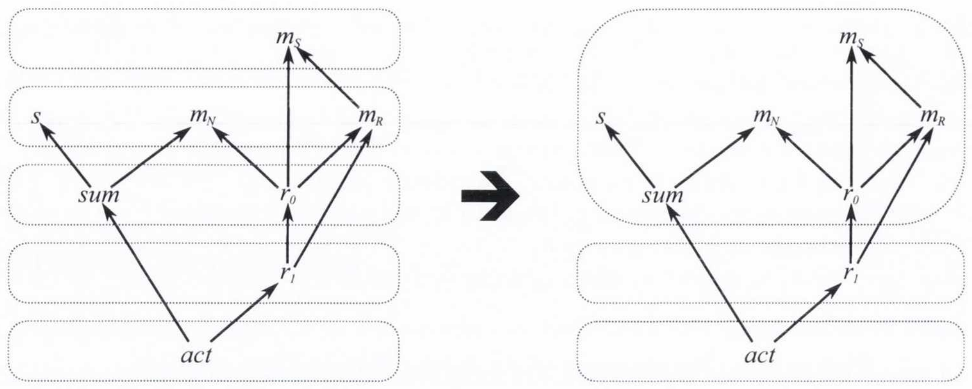


Figure 4.4: Example scheduling of a data dependency graph.

following Sosna Stream Core program fragment.

```
def $m_N$ = ... in
  def $sum$ = staggr + $s$ $s$ $m_N$ $m_N$ 0 0 in
    def $m_S$ = ... in
      def $m_R$ = tuple2 (extern $id$) ($\#_2^2$ $m_S$) in
        def $r_0$ = staggr $max$ $s$ $s$ $m_R$ $m_S$ 1 2 in
          def $r_1$ = staggr $max$ 0 $r_0$ $m_R$ $m_S$ 0 1 in
            output $act$ smap / $r_1$ $sum$;
```

The program performs two data aggregation tasks: nodes sum the values of the s stream across their neighbourhoods (sum) and the maximum value of s is selected among nodes that are members of a two-hop wide dynamic cluster topology (m_S and m_R). The result of program evaluation is present at the cluster topology's root nodes and it is equal to the maximum value of s divided by the local value of sum. Figure 4.4 presents the data dependency graph for this program. Note that the evaluation order of expressions located at the same *level* in the graph can be arbitrary if we assume that all expressions from the level above it have been already evaluated.

The scheduling algorithm is concerned with partitioning the set of all program identifiers and their associated expressions into a bounded number of subsets that correspond to a certain partitioning of the dependency graph and which we refer to as communication *rounds*. The communication schedule, therefore, is a sequence of communication rounds all of which, with the exception of the first and the last one, containing at least one `staggr` expression. Different `staggr` expressions assigned to the same communication round are executed in parallel and the scheduling algorithm ensures that communication rounds cannot contain two `staggr` expressions that are dependent on each other. As a result, the duration of every communication round is exactly that of a single round of neighbourhood data aggregation and the duration of the whole schedule is a multiple of this value. Figure 4.4 shows the relation between a program's data dependency graph and its scheduling, the dashed boxes represent the composition of different communication rounds. Note that, although there cannot be dependent `staggr` expressions in the same round, dependencies among other expressions are allowed. This is because these expressions do not involve communication, hence their evaluation should take significantly less time than one neighbourhood data aggregation round. In the rest of the presentation we refer to these expressions as *instant*.

The scheduling algorithm requires the program to be simplified by un-nesting all nested `init`, `def` and `set_after` expressions. The transformation is simple and for the above example program it results in the following sequence of definitions that we refer to as schedule entries.

$$\begin{array}{l}
(\texttt{def}, s, \ldots), \\
(\texttt{def}, m_N, \ldots),\ (\texttt{def}, sum, \texttt{staggr} + s\ s\ m_N\ m_N\ 0\ 0), \\
(\texttt{def}, m_S, \ldots),\ (\texttt{def}, m_R, \texttt{tuple2}\ (\texttt{extern}\ id)\ (\#^2_2\ m_S)) \\
(\texttt{def}, r_0, \texttt{staggr}\ max\ s\ s\ m_R\ m_S\ 1\ 2) \\
(\texttt{def}, r_1, \texttt{staggr}\ max\ 0\ r_1\ m_R\ m_S\ 0\ 1) \\
(\texttt{output}, act, \texttt{smap}\ /\ r_1\ sum)
\end{array}$$

Each schedule entry, therefore, comprises three elements: type, which can take one of four different values: `init`, `def`, `set_after` and `output`, program identifier, and the corresponding expression. For each schedule entry $e = (type, id, expr)$ we denote the set of program identifiers referenced in $expr$ as $depset(e)$ and we define the following function:

$$kind\ (type, id, expr) = \begin{cases} \texttt{init} & \Longleftrightarrow \quad type = \texttt{init} \\ \texttt{instant} & \Longleftrightarrow \quad type \in \{\texttt{def},\ \texttt{set\_after}\} \\ \texttt{async} & \Longleftrightarrow \quad expr = \texttt{staggr}\ \ldots \\ \texttt{output} & \Longleftrightarrow \quad type = \texttt{output} \end{cases}$$

Additionally, we define the relation $\prec$ on entry kinds to be:

$$\texttt{init} \prec \texttt{instant} \prec \texttt{async} \prec \texttt{output}$$

The algorithm operates in two phases: first it constructs an approximation of the final schedule by building a list of schedule levels each of which being a set of schedule entries of the same kind. We extend the notion of entry kinds to schedule levels by saying that a level L has a kind k when $\forall e \in L.(kind\ e = k)$ and, additionally, denote the set of identifiers of all entries $e \in L$ as $(ids\ L)$. In the second phase, certain sequences of schedule levels are merged together in order to form schedule rounds. Pseudo-code for both phases is presented in Algorithm 4.1.

Algorithm 4.1 The communication scheduling algorithm for the Sosna Stream Core language.

```
def $independent\ e\ level = ((depset\ e) \cap (ids\ level)) = \emptyset)$

def $insert\ e\ [\,] = \{e\}$
  | $insert\ e\ level :: schedule =$
  if $independent\ e\ level$ then
      if $(insert\ e\ schedule) \neq$ false then
          return $level :: (insert\ e\ schedule)$
      else
      if $(kind\ e) = (kind\ level)$ then
          return $level \cup \{e\} :: schedule$
      else
      if $(kind\ e) \prec (kind\ level)$ then
          return $level :: \{e\} :: schedule$
      else
          return false
  else
      return false

def $partition\ entryList =$
  $schedule := [\,]$
  foreach $e \in entryList$ do
      if $(insert\ e\ schedule) \neq$ false then
          $schedule := (insert\ e\ schedule)$
      else
          $schedule := \{e\} :: schedule$
  return $reverse(schedule)$

def $merge$ $\mathtt{instant}_1 :: \mathtt{instant}_2 :: rest = merge\ (\mathtt{instant}_1 \oplus \mathtt{instant}_2) :: rest$
  | $merge$ $\mathtt{instant} :: \mathtt{async} :: rest = merge\ (\mathtt{instant} \oplus \mathtt{async}) :: rest$
  | $merge$ $\mathtt{instant} :: \mathtt{output} :: [\,] = \mathtt{instant} \oplus \mathtt{output})$
```

The first phase (the *partition* function) starts with the original schedule entry sequence and with an empty list as the initial schedule level sequence (we use the *head* :: *tail* notation to distinguish between a list's head element and all remaining elements). The schedule level sequence corresponds to the reversed schedule, hence, the first level contains entries which would be executed at the end of the program's cycle. The individual entries are inserted to the level list, one by one, by selecting the level possibly the closest to the end of the list. The entries are moving on through the schedule level list as long as their identifiers are independent of any identifiers in the given level (the *independent* function). Once a conflict is found, the entry is either added to the preceding level or, when it has a smaller kind, it is added to a newly created level located between the current and the preceding levels. If an entry cannot be added to an existing level list, a new level added at the beginning of the list, in order to accommodate it.

The first phase groups all `init` entries in the first schedule level and all `output` entries in the last schedule level because these entries are independent by definition. The resulting level sequence corresponds to a partitioning of the data dependency graph into a sequence of sets each of which containing independent entries of the same kind. Also, for any two levels L_i and L_j such that $i < j$, we have:

$$\forall e \in L_i.(\mathit{independent}\ e\ L_j)$$

The second phase is used as a post-processing stage and it simply merges all consecutive `instant` levels together, as well as, it merges all consecutive pairs of `instant` and `async`, and `instant` and `output` levels. The symbol $\oplus$ represents merging of sequences and it corresponds to the fact that the ordering of entries becomes important as the merged levels are likely to contain dependent expressions. As a result of this grouping, the first set of entries comprises only `init` entries (if present in the program), the last one comprises only `instant` and `output` entries, and all other sets comprise at least one `async` entry. The first sequence, therefore, becomes the *program initialisation sequence* and the last one constitutes the *actuation round* while each of the remaining sequences becomes a single communication schedule round. The sequence of all communication schedule rounds followed by the actuation round constitutes the basic communication schedule which might be extended in the the target platform code generation stages with additional delays, e.g., sleep periods.

4.1.11 Send/Receive Transformation

The basic communication schedule is a sequence of communication rounds each of which comprising a number of `staggr` expressions. Each `staggr` expression represents a neighbourhood stream aggregation round in which every node sends at most one data item. Scheduling several neighbourhood stream aggregation rounds in parallel can be done by putting all data items a node wants to send within the given communication round into a single message which is then decomposed by all of its receivers that extract the data items of interest. As a result of this approach, SOSNA applications send at most one radio message in every communication round (when we assume that all required data items can be fit in a single message).

The send/receive transformation stage converts all `staggr` expressions into the corresponding abstract message sending expressions and into a set of data item receive expressions representing aggregation of the incoming data items at the receiver side. We use the example program presented at the beginning of the previous section to demonstrate the idea. The program is scheduled to comprise three schedule rounds that correspond to the scheduling presented in Figure 4.1. We use schedule entry identifiers simplicity:

$$[m_S, s, m_N, m_R, sum, r_0,],\ [r_1],\ [act]$$

The transformation introduces the following two expressions that assign messages unique identifiers and define them in terms of the data items they comprise, as well, as in terms of the associated topologies:

$$\texttt{send}\ m_0 = [(s, m_N)\ (s, m_S)]$$
$$\texttt{send}\ m_1 = [(r_0, m_S)]$$

Topology descriptors need to accompany the data items because different data items might be destined for different receivers. The rule of thumb is the following: each data item is represented in the message by its value and by a network identifier which, in the case of neighbourhood topologies is equal to `bcast`, in the case of aggregation over a cluster topology it equals to the parent node's identifier while in the case of propagation over cluster topologies it represents the sender's identifier so that the receiving nodes can accept messages only from their tree parents. In all three cases, the destination topology descriptor provides all necessary information. Additionally, in order to facilitate the subsequent program partitioning stage, different messages are composed for nodes of different device classes so that each message contains only those data items that originate in the given round from nodes of the same device class .

While the `send` expressions define the sending part of the `staggr` operator, the `receive` expressions define data item reception and aggregation. For each individual `staggr` expression a separate `receive` expression is generated and they take the following general form.

$$\texttt{receive}\ m\ f\ i\ n\ r\ l_R\ result_{id}$$

Following the notation introduced in Section 4.1.7, the parameters are respectively: the identifier of the accepted message, the aggregation function, the initialisation stream, the index of the received data item within the message, the receiver's topology descriptor, the receiving level and the identifier of the stream that will be the result of this aggregation step. The following expressions are generated for the above example program.

`receive` $m_0\ +\ s\ 0\ m_N\ 0\ sum$

`receive` $m_0\ max\ s\ 1\ m_R\ 1\ r_0$

`receive` $m_1\ max\ 0\ 0\ m_R\ 0\ r_1$

This information is sufficient for every node to know which messages it should receive. Also, at this stage, programs contain no expressions whose semantics spans two different device classes, therefore, they can be safely partitioned.

4.1.12 Program Partitioning

Knowing the types and device classes of all program expressions and having transformed the language so that all expressions have local semantics, the compiler can partition the global program into a set of device class-specific sub-programs. The task is relatively simple and involves running the same program filtering algorithm several times, each time parametrising it with a different device class identifier.

The filtering is performed at the level of individual schedule entries because they enjoy a convenient property that their expressions are evaluated entirely at a single device class. Because all language's constructs have local semantics, no expression can directly refer a program identifier that is evaluated at a different device class. Therefore, for each device class, it is safe to remove from the global program all schedule entries that are annotated for evaluation at all other device classes.

The program partitioning stage performs additionally program translation from the scheduler's representation to the the SOSNA Stream Local language that defines programs as sequences of schedule rounds comprising sequences of basic program statements. The transformation stage outputs a set of localised programs each of which to be executed by devices of a different class.

4.1.13 SOSNA Stream Local

The SOSNA Stream Local language serves as an intermediate program representation language that is used by various compiler backends as the basis for generation of platform-specific code. SOSNA Stream Local is an imperative language whose semantics are close to how individual devices would operate but at the same time they do not involve platform-specific details. Figure 4.5 presents the grammar of the language.

$$\begin{array}{lll}
f & ::= & \texttt{tuple2} \mid \#^2_1 \mid \#^2_2 \mid + \mid - \mid \times \mid / \mid \ldots \\
c & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid 2 \mid \ldots \\
\mathit{fun} & ::= & \lambda x.e_f \mid \texttt{rec}\ x.\lambda x.e_f \\
\mathit{program} & ::= & \texttt{functions}\ (x = \mathit{fun})^*\ \texttt{init}\ (x = (c \mid \texttt{nil}))^*\ (\texttt{round}\ \mathit{stat}^*)^+ \\
\mathit{stat} & ::= & \texttt{def}\ x = e_s; \mid \texttt{set}\ x = e_s;\ \mid \texttt{output}\ x\ e_s; \mid \\
 & & \texttt{send}\ x = [(e_s, e_s)^+] \mid \texttt{receive}\ x\ f\ i\ n\ r\ l_R\ x \\
e_s & ::= & c \mid f \mid x \mid \texttt{extern}\ x \mid \texttt{input}\ x \mid \texttt{vals}\ e_s \mid \texttt{time}\ e_s \mid \\
 & & \texttt{smap}\ x\ e_s \mid \texttt{merge}\ e_s\ e_s \mid \texttt{within}\ e_s\ e_s \\
e_f & ::= & c \mid x \mid x\ e_f \mid \texttt{if}\ e_f\ \texttt{then}\ e_f\ \texttt{else}\ e_f \mid \texttt{def}\ x = e_f\ \texttt{in}\ e_f
\end{array}$$

Figure 4.5: The grammar of the SOSNA Stream Local language.

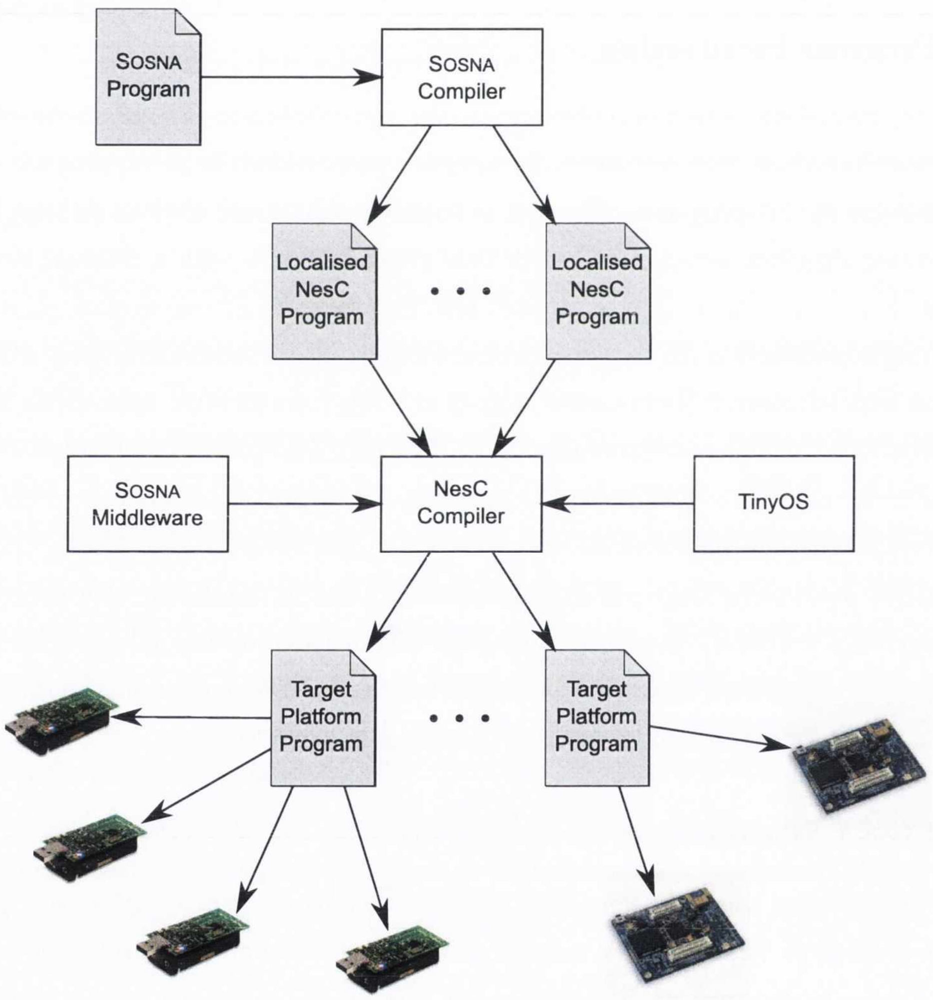


Figure 4.6: The SOSNA compilation framework.

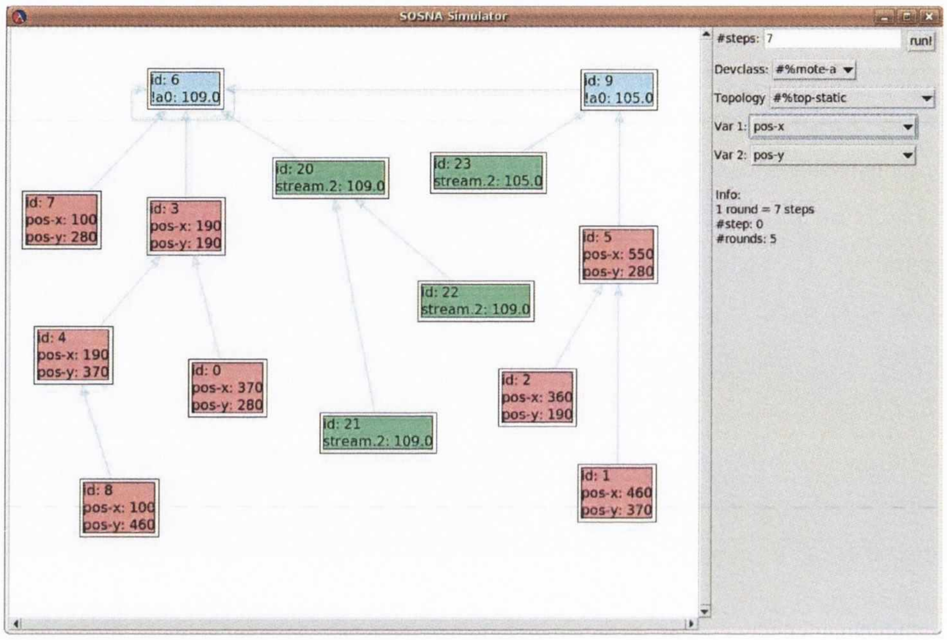
The language structures programs as sequences of rounds that comprise possibly empty sequences of basic program statements. The statements are either stream expression definitions, stream identifier value update statements, output (actuation) statements, message send or reception statements. The scope of stream expression definitions is confined only to the round they are defined in allowing this way for memory reuse across different rounds. All stream identifiers declared in the `init` section are considered global, i.e., the memory they occupy cannot be freed. The `init` identifiers, in addition to all `fby` buffers (see Section 4.1.6) include also those stream definitions whose scope cannot be confined to a single round and they include, for example, the identifiers storing neighbourhood stream aggregation results (the `nil` value is used for their initialisation). Also, note that the language separates functions from stream expressions and explicitly assigns them names so that they can be referred to only through their identifiers.

SOSNA Stream Local is intended as the input language for various compiler backends. The existing compiler implementation provides two backends that are the SOSNA Simulator and the SOSNA Machine language (SMachine). The former translates SOSNA Stream Local programs into Scheme classes that are instantiated by the simulator to create objects representing SOSNA programs running on individual network nodes. The later is a virtual machine-like language that serves as a basis for efficient translations to C-based languages. The following two sections provide more detail on each of the backends and Figure 4.6 describes the overall SOSNA compilation process that involves an external compiler for the compilation of NesC-based translations of the SMachine target.

4.1.14 The SOSNA Simulator

The simulator allows to run SOSNA programs after they are compiled to a Scheme class whose interface comprises methods for creating objects representing program instances, as well as, for controlling program execution in terms of triggering the execution of individual schedule rounds. The current implementation of the simulator accepts as input:

- a graph that represents wireless network connectivity,
- a mapping of device class identifiers to graph nodes,
- a class representing the translated Sosna program that is used to create program instances for every node in the graph and to connect these objects according to the topology of the graph,
- an environment model that generates sensor readings for all network nodes (program instance objects) and which receives and accommodates all actuation signals coming from these nodes.



#steps: 7
Devclass: #%mote-a
Topology #%top-static
Var 1: pos-x
Var 2: pos-y

Info:
1 round = 7 steps
#step: 0
#rounds: 5

Figure 4.7: The graphical user interface of the SOSNA Simulator.

Once the simulator constructs the network model, it sends to all program instances an initialisation signal that results in the execution of the `init` block described in Section 4.1.13. The simulator operates in a synchronous manner executing one application's schedule round at a time. Before each round commences, the simulator generates a random permutation of all network node identifiers and sends a new-round signal to the nodes following this ordering. Internally, each round is split into two parts of which the first comprises the computation of the round's instant expressions and message sending while the second one involves processing of the received messages so that the results of data aggregation can be computed. The simulator employs a simplified channel model that currently deterministically (although in random order) delivers all messages to their destinations. Future versions of the simulator might extend this model by employing more realistic models of wireless signal propagation.

The simulator in its current form is intended for early design stages and it includes a graphical user interface (GUI) that allows to inspect the current values of program stream identifiers, as well as, it is capable of visualising a chosen topology descriptor. The simulator's GUI is presented in Figure 4.7.

4.1.15 SOSNA Machine

The SMachine language deals primarily with the evaluation of stream expressions as defined by the e_s stream expression class in the SOSNA Stream Local grammar (Figure 4.5). Apart from the representation of stream expressions, the language has a similar structure to SOSNA Stream Local which assumes partitioning the program into a sequence of schedule rounds. The main motivation for the language is the fact that the representation of absent stream values can be costly because every single stream operation involves checking if its arguments are not null. The language tries to avoid annotation of every stream expression with `if/then/else` statements and employs a general model in which stream operator arguments are stored as pointers on a stack so that value presence checking comes down to checking if the pointers are null. We focus, therefore, on the description of the core ideas underlying the language and on the presentation of its instruction set.

The language defines an abstract machine that comprises a heap for storing allocated stream values and a stack that stores pointers to these values. SMachine instructions allocate, free and modify the heap memory leaving pointers to their results on the stack. The core characteristic of the language is the ability of the compiler to figure out the worst-case heap and stack sizes required by the program because the language does not include looping constructs. This is an important property when building WSN/WSAN applications because memory resources are likely to be scarce, as well as, because bounding an application's memory consumption at compile time improves its reliability.

Similarly to SOSNA Stream Local, SMachine programs start with the `init` section that allocates heap memory for all global stream identifiers while saving pointers to these memory blocks on the stack. Subsequent language instructions cannot release this memory and either allocate and free additional blocks or push references to already allocated memory blocks to the stack. This situation is depicted in Figure 4.8 in which the gray boxes represent globally allocated memory blocks and their corresponding pointers. We present the full instruction set of the language in Appendix A and in this section we confine ourselves only to the presentation the translation of an example SOSNA Stream Local program fragment.

In Section 4.1.13 we noted that `def` statements confine the scope of their definitions to the enclosing schedule rounds. This is done by freeing the memory allocated by such statements at the end of the round. Consider the following definition:

`def` $x =$ `within` e_1 e_2

$\ldots$ //other statements

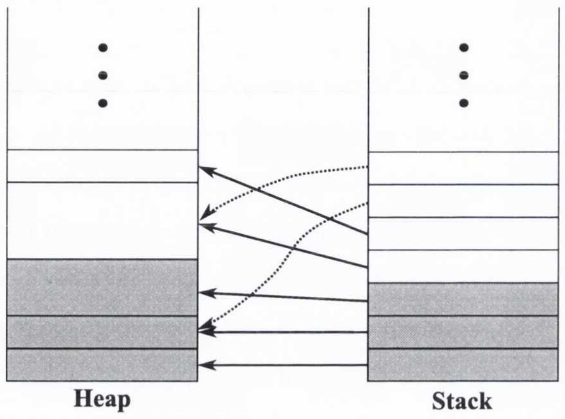


Figure 4.8: The relation between the SMachine's heap and stack.

If we assume that x is a stream of type Int, then corresponding translation to SMachine is the following:

```
alloc Int
...//"within e1 e2"
...//other statements
free Int
```

In order to abstract away the details of low-level data representation, SOSNA Machine uses types to denote sizes of memory blocks and, as a result, the maximum program heap size. After SMachine programs are compiled to a particular target platform such as, for example, NesC or gcc C, The types are replaced with integer numbers representing type sizes in platform-specific units.

Stream expressions are evaluated in SMachine in a similar way as the ordinary arithmetical expressions are in other machine languages: pointers to arguments of SMachine operations are left on the stack and then they are replaced with the pointer to the operation's result. All stream operations are allowed to allocate any amount of heap memory but they must finish execution with exactly one pointer allocated on the stack. Although this mode of operation makes the stack volume easy to track, the amount of allocated heap memory might depend on the values streams take at runtime. Consider the (`within` e_1 e_2) expression[4]: the evaluation of e_2 should not be performed when e_1 is either absent or `false`. In such case, only the worst-case memory consumption can be computed when we assume that both e_1 and e_2 are evaluated. In order to cope with such situation all SMachine uses two instructions `saveHeap` and `restoreHeap`. The above example would be translated to the

[4]This is a SOSNA Stream Local expression, hence, both e_1 and e_2 may not comprise communication operators.

following code.

```
alloc Int
saveHeap
...//"within $e_1$ $e_2$"
restoreHeap
...//other statements
free Int
```

The `saveHeap` instruction pushes to the stack a pointer to the current top of the heap while `restoreHeap` sets the top of the heap to be memory location pointed to by the pointer at the top of the stack.

If we assume, for simplicity, that $e_1 =$ `true` and that $e_2 = y$ then the above program would be translated to:

```
alloc Int
saveHeap
...//"$e_1$"
present? 1
ifTrue
    ifTrue
        ...//"$e_2$"
    else
        pushNull
else
    nop
restoreHeap
...//other statements
free Int
```

The `within` expression is evaluated as follows: first, the first argument is evaluated and a pointer to its value is left on the stack. The `present?` instruction consumes this pointer and pushes a new pointer to a Boolean value that is equal to `true` if the previous top pointer was not null, and that is equal to `false` otherwise. The `ifTrue/else` statement is a form of a controlled jump instruction and it executes its first branch if the top pointer (left by `present?`) points to `true` and it executes the second branch otherwise. Therefore, the first branch of the first `ifTrue` is executed when e_1 evaluated to a non-null value and the program proceeds to the evaluation of e_2 when this value was equal to `true`. In the case when e_1 was `false` a null pointer is pushed to the stack as the final result, as required by the operator's semantics. In the case when e_1 was null, nothing needs to be done, as this is also the final result of the expression.

The above example shows how structured stream expressions are evaluated in SOSNA Machine, as well as, it shows how memory is managed by the language. Sosna Machine instructions are directly translated to pointer manipulating C expressions that operate on two statically-allocated arrays representing the stack and the heap. This translation has been implemented for the NesC programming language and is the basis of the NesC target of the SOSNA compiler.

4.2 Execution in Mote Networks

SOSNA applications can be executed in wireless networks of TinyOS-supported devices called the motes. TinyOS is an open source operating system designed for use on highly resource constrained devices and written in the NesC programming language (see Section 2.1.4 for an overview). TinyOS offers a very minimal model in which applications are compiled against the operating system's (OS) source code to form a monolithic program that includes both the application code and the operating system. TinyOS applications are specified as graphs of asynchronously communicating components and the operating system does not support preemption, hence, concurrency has to be manually managed by the programmer. The system uses the concept of a *task* to represent a piece of computation that is executed in its entirety and whose execution can be delayed allowing for a way of breaking computationally-heavy procedures into fragments in order not to block the operation of other components. TinyOS uses a static memory model, i.e., dynamic memory allocation is not possible unless implemented by hand.

The most recent version of the operating system - TinyOS2, supports several hardware platforms most of which have severely constrained resources. Table 4.1 compares the some characteristics of four popular mote platforms supported by the OS: MicaZ, Iris, TelosB and Imote2, all available from Crossbow Technology (2009). The devices, with the exception of Imote2 have very little RAM, limited program memory (i.e., non-volatile memory for storing the application code) and relatively slow processors. The table shows that RAM is the most constraining factor requiring compilers to focus primarily on optimising its use.

A variety of sensors are available for the mote platforms but mainly in form of external sensor boards that can be attached the motes through specialised connectors. The available sensors include light, humidity, temperature, acceleration, Earth's magnetic field, image (video) and sound sensors, and TinyOS provides unified interfaces for controlling their operation.

In the following sections we describe how SOSNA applications are executed in mote networks. First

	MicaZ	Iris	TelosB	Imote2
Processor Clock	8MHz	16MHz	8MHz	3-416MHz
RAM	4kB	8kB	10kB	256kB
Program Memory	128kB	128kB	48kB	32MB
Flash Memory	512kB	512kB	1MB	32MB

Table 4.1: Comparison of hardware capabilities of chosen TinyOS-supported mote platforms.

we overview the Flooding Time Synchronisation Protocol (FTSP) that underpins the synchronised operation of the whole network. Then we describe the SOSNA Middleware, which is a library building on the FTSP protocol and driving the operation of SOSNA applications. We conclude the section with a discussion of how duty cycled is performed in mote networks.

4.2.1 Time Synchronisation

The Flooding Time Synchronisation Protocol (Maróti et al. 2004) is a relatively simple time synchronisation protocol that targets WSN applications and that employs low-level radio message time-stamping in order to reduce the amount of communication needed to achieve synchronisation. The protocol operates by propagating a flooding wave of time synchronisation messages away from the time synchronisation root. Each of the time synchronisation messages carries a time stamp that represents the sender's local time recorded at the time when the message was being transmitted to the wireless channel, bypassing this way much of the unpredictable delays introduced by the OS message processing. The nodes store time synchronisation message reception times in order to fit a linear regression model allowing them to estimate their clock drift and compensate for it in the global time estimation.

The protocol operates in a periodic manner regularly refreshing network time synchronisation. In addition to the flooding wave propagation, the protocol also provides for time synchronisation root election so that, on one hand, it can start operation without explicit configuration, and on the other, it can tolerate node failures by choosing a new leader when the previous one stops responding. This is done by counting at each node the number of time synchronisation refreshes (heartbeats) originating from the current root. After a node notices that the number of missing heartbeats exceeds a threshold it elects itself to be the new root and through a simple contention mechanism, similar to the SOSNA's leader election process, the new synchronisation root is selected to be the node with the lowest network identifier.

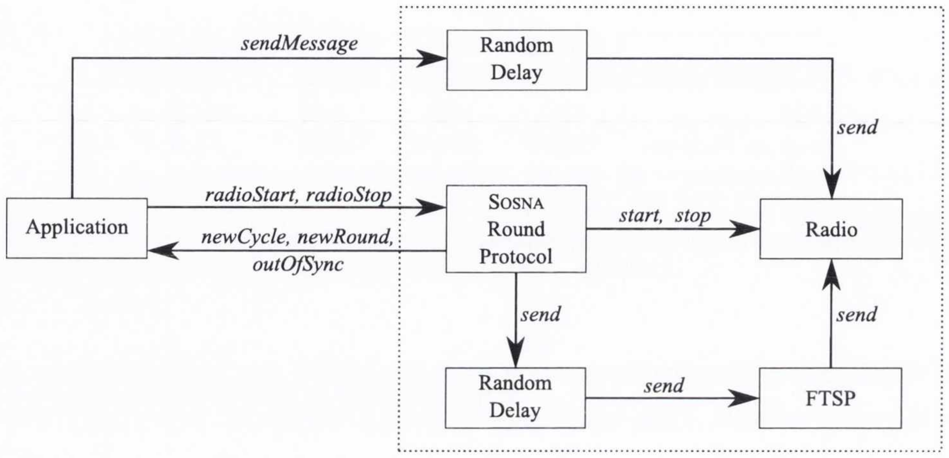


Figure 4.9: The SOSNA Middleware.

4.2.2 The SOSNA Middleware

The SOSNA compiler delegates the low-level aspects of application operation to a runtime support middleware that is a collection of TinyOS components whose tasks are:

- to provide time synchronisation in the network,
- to start the application when time synchronisation is established and to stop it when time synchronisation is lost,
- to trigger execution of subsequent application schedule rounds,
- to handle sending of SOSNA messages.

The middleware comprises three core components that are: a modified version of the FTSP protocol, the SOSNA Round Protocol (SRP) that coordinates the operation of FSTP and triggers application round execution, and a message send component whose task is to send a message after introducing a random delay. Additionally, SOSNA applications require access to a radio control interface for switching the radio on and off. We use for this purpose the *SplitControl* interface which is implemented by default be the TinyOS communication module. Figure 4.9 shows the inter-dependencies among these components and the application code.

The SOSNA middleware operates according to the following general application execution model. The application cycle starts with a time slot that we refer to as the *service round* and that is reserved for the operation of the time synchronisation protocol, as well as, for sensor sampling. The current implementation, for simplicity, assumes the service round to have the same duration as the application

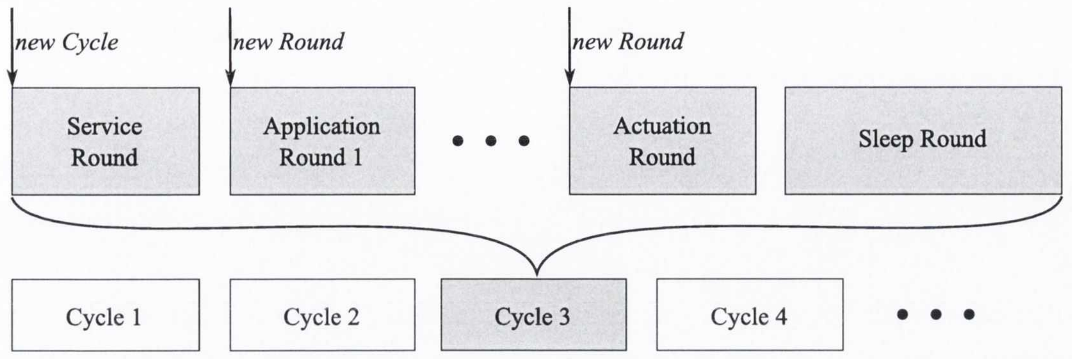


Figure 4.10: The general application time schedule.

rounds but this does not have to be the case in general. The service round is followed by a sequence of application rounds, of which the last one is the actuation round, as described in Section 4.1.13. The actuation round is followed by a global sleep round whose duration is a compilation parameter and can be arbitrary (including the duration of zero seconds). The duration of one application round is also a compilation parameter and its value has to be selected in accordance with the application's requirements, in particular, it cannot be too short so that each node has enough time to send a message and to receive messages from all of its neighbours whose maximum number is specific to a particular deployment topology. There is no technical reason for limiting the upper bound on round duration other than application responsiveness because longer rounds result in longer application cycles. Figure 4.10 presents the overall composition of the application cycle.

The main task of the SOSNA Round Protocol is to control the propagation of the time synchronisation wave generated by FTSP, to detect and to react accordingly when FTSP reports having lost synchronisation with the time source and to notify the application when a new cycle and a new round should begin. When the application is started it first initialises SRP with the following three parameters: the duration of one application round, the total number of rounds (including the service round) and the duration of the sleep period. As a result, the protocol switches the radio on and starts FTSP in order to synchronise the network for the first time. Once FTSP reports that the time is synchronised, SRP uses the current value of the global time to calculate the starting time of the nearest application cycle. This is done according to the following simple formula (D_{cycle} represents the total application cycle duration, and T_{global} stands for the current value of the global time):

$$T_{start} = \left\lfloor \frac{T_{global}}{D_{cycle}} + 1 \right\rfloor \times D_{cycle}$$

SRP then sets a local timer to fire when T_{start} elapses and sends to the application the *newCycle* signal. After that, SRP sets the timer to fire when the next cycle begins and, at the same time, it starts another timer to fire when each of this cycle's rounds commences so that the application can be sent the *newRound* signal (see Figure 4.10).

Although the duration of application rounds is a compilation parameter, care must be taken to ensure that it is not too short, i.e., that there is enough time left to perform all computations and communications scheduled in each round. The current prototype Sosna compiler leaves this entirely up to the programmer allowing for arbitrary durations of application cycles. Hence, if some computations take too long (e.g., an erroneous, unbounded-depth recursive function call), nothing can stop one round from running into another due to the lack of preemption in TinyOS. However, as we noted in Section 3.3.4, a fully-fledged compiler should employ some means to estimate the execution time of program's scalar expressions and this seems to be within reach of the existing compiler technology. Alternatively, it is possible to come up with a reasonable choice for application round duration through an estimation of network topology density and communication throughput, as well as through analysis and profiling of function execution times.

At the beginning of each new cycle SRP uses FTSP to obtain the current value of the global time. When FTSP loses synchronisation with the root, it returns a null value, hence, SRP can reset the application by sending the *outOfSync* signal. The application then resets the heap and stack pointers and it invokes the initialisation procedure described in Section 4.1.13. At the same time, SRP moves to the initial state when the radio is switched on and synchronisation messages are being awaited. In the following sections we describe the system operation during the four main round types: the service round, the application round, the actuation round and the sleep round.

The Service Round

The service round is designed to accommodate time for sampling the sensors and for the operation of the time synchronisation protocol. Sensor sampling is performed by the application that requests sensor readings for all program inputs as soon as it receives the *newCycle* signal. TinyOS uses an asynchronous sensor sampling interface that requires that, first, sensor readings are requested and then the values are delivered by means of a call-back function. The Sosna compiler declares a similar interface but which, in contrast to its TinyOS counterpart, additionally requires sensor drivers to provide the real time at which the sensor reading was obtained. The reason for it is such that program inputs might represent higher-level sensors that report only event detection values in which case the detection time might be different than, for example, the beginning of the cycle. This property of program inputs is explained in Section 3.2.2 and in Figure 3.6 and below we present the generic NesC interface for for Sosna's sensor drivers.

```
interface Input<dataTyype> {
    command error_t read();
    event void readDone(error_t status, dataType data, gtime_t localTime);
}
```

Before a sensor reading is requested, the application clears the global pointer that points to its current value indicating this way that the input is null in this instant. This value is then overwritten when the sensor reading is reported by the sensor driver. In contrast to TinyOS, SOSNA breaks the connection between sensor value request and the value delivery report allowing this way null input values.

While the application is reading sensor values, SRP is controlling the FTSP protocol. The protocol's low-communication overhead makes it ideally suited for use in SOSNA applications because it requires every node to send exactly one synchronisation message per one time synchronisation wave. SRP exploits this fact and schedules the protocol message to be sent in the service round. However, in order to minimise the overhead of maintaining the static cluster topology, the SOSNA Middleware augments the protocol by extending its messages to carry, in addition to time synchronisation-related information, two fields that denote the distance in hops to the global time synchronisation source and the distance to the node's static cluster head. This way, the static cluster topology is constructed and refreshed each time the time synchronisation wave is propagated.

Because it would be impractical to assume that the whole time synchronisation wave should spread within the service round, the middleware splits the propagation across several application cycles. The procedure requires that the network is already synchronised and that network nodes know both the global time and their distance to the time source, as indicated by the static topology. First, the total application cycle number is computed as:

$$n = \left\lfloor \frac{T_{global} + \epsilon}{D_{cycle}} \right\rfloor$$

The ϵ represents a small correction that is added to account for possible timer inaccuracies in situations when the new cycle starts slightly earlier. A node sends the FTSP message when:

$$n \textbf{ modulo } L_{max} = L_n \textbf{ modulo } L_{max}$$

where L_n is the the node's distance to the time source in the n-th application cycle and L_{max} is a compilation parameter defining the tree refresh period. As a result, the time synchronisation wave gradually spreads one synchronisation tree level per application cycle. The L_{max} parameter can be set to correspond to the maximum time synchronisation tree depth, in which case a new time synchronisation wave starts as soon as the previous one finishes, or it can be set to smaller or larger value to respectively decrease or increase the refresh rate.

The above condition is evaluated at the beginning of every service round and once it holds, SRP instructs the FTSP protocol to send the synchronisation message. Before this happens, however, SRP introduces a random delay $d \in [0, D_{round}/3]$, where D_{round} is a compilation parameter denoting the duration of one round of neighbourhood communication. This is a practical requirement, as we noticed during the evaluation, that due to the synchronised times of message sending, channel collisions occur causing loss of synchronisation messages and resulting in nodes loosing synchronisation and restarting. A possible explaination of this fact is the potential coupling among neighbouring nodes' random number generators due to a faulty CSMA MAC protocol implementation. The bug causes the motes to start their transmissions at the same times because they repeatedly choose exactly the same transmission backoff intervals. This leads eventually to the failure of message transmission. As Chapter 5 shows, this approach significantly improves communication reliability.

The Application Round

Application rounds comprise both evaluation of instant expressions, as well as, sending and reception of application messages. SOSNA's semantics state that in every application round network nodes may send at most one message and may receive unbounded number of messages. Two practical issues arise when implementations are considered. Firstly, the application round duration has to be chosen to be long enough to allow for the worst-case channel contention which is related the maximum degree of the network connectivity graph. This value is application specific and should be established either by empirical means or by constraining the connectivity graph in some predictable way. Secondly, since wireless data packets packets have bounded capacity applications whose abstract messages cannot fit a single data packet either have to be rejected by the compiler or their transmission has to be split into several packets. Although the current implementation of the compiler takes the former approach, there seems to be little technical difficulty in extending it to handle more sophisticated transmission modes. Thanks to the single message-per-round policy, SOSNA applications allocate memory for only a single message buffer that is shared across all application rounds.

All application rounds follow the same pattern: first the message buffer is cleared, then all instant expressions are evaluated (including loading data items to the message buffer) then, if the message is not empty, it is sent out and, finally, if there are any data items expected to be received, appropriate data item handlers are added to the handler list for invocation upon neighbour message reception. Similarly to the time synchronisation messages, a random delay $d \in [0, D_{round}/3]$ is introduced in order to avoid channel collisions resulting from synchronised sending.

As we said in Section 4.1.11, messages are sequences of data items and their associated destination topology descriptors. The minimum information that needs to be transmitted is the data item's value and the destination network identifier comprising the destination topology descriptor. TinyOS uses 16-bit network identifiers but the current compiler implementation splits them into 8-bit device class descriptors (the most-significant bits) and 8-bit node identifiers (the least significant bits). Network nodes are, therefore, uniquely identified by providing the full 16-bit address and can be distinguished within their device classes based on their 8-bit node identifiers. Because, the message identifier and the data item's index in the message uniquely determine the sender's and the receiver's device classes, data items can be accompanied by only 8-bit destination addresses.

The current compiler implementation assigns a fixed index to every data item in a message. Bit fields are used to represent data value presence and absence in the given instant. In order to save space, all presence fields are grouped in a single 16-bit integer field that limits the maximum number of data items per message to sixteen. This value is platform specific as different radios and MAC protocols might define different message sizes. For example, the TinyOS defines the the maximum message payload size for the TelosB platform to be 118 bytes while for the Mica2 platform to be 28 bytes. The compiler can use this information to allow larger messages on more capable platforms.

SOSNA messages are mapped onto NesC structures and unions. Every message contains the 8-bit message identifier and the 16-bit data item presence field that is followed by a sequence of pairs of data items and their 8-bit destination identifiers.

The Actuation Round

Actuation rounds may comprise instant stream expressions that correspond to the values of the program outputs. Once computed, these values are passed on to the actuators. Similarly to sensor drivers, SOSNA specifies a TinyOS interface that needs to be implemented by all actuator drivers. The following code presents the interface.

```
interface Output<valType> {
    command void write(valType val);
}
```

Contrary to the `Input` interface, `Output` does not declare a call-back function because the SOSNA's semantics assume that outputs return no values. It can be imagined, however, that actuation results such as, for example, the error code of the last actuator invocation can be useful to the programmer. Such functionality can be achieved by defining a virtual input driver that instead of a sensor value gives the result of actuation in the previous instant. Indeed, this is how we implement in Chapter 5 actuation time measurement - actuator drivers simply log the time of the `write` command and return it in the next instant as a virtual sensor reading.

Actuator synchronisation is an important feature of the language. However, because we allow instant stream expressions to be evaluated before the actuators are instructed it is possible that these computations might affect the real time of actuation. We see two possible solutions to this problem. On one hand, if recursive functions are not used, the actual delays should be minimal because they involve the execution of a fixed number of arithmetical operations. On the other hand, if higher actuation synchronisation was required, the actuation round could be split in two parts of which the first one would be reserved for the computation while the second one would only trigger the calls to actuator drivers. The choice of any of these options does not affect the language's semantics and, in fact, is implementation specific. Therefore, for simplicity, the prototype compiler implements the first approach.

The Sleep Round

The sleep round is reserved for power conservation in low-rate applications. It is a compile time-specified time period that does not involve any activities. The compiler can, therefore, generate code that switches the radio off and puts the processor to a low-power mode. Because TinyOS2 automatically manages the processor's power state, the only thing the application need to do is to switch the radio transceiver off. This, however, happens usually before the sleeping round commences and the following section discusses the details of the implemented radio duty-cycling policies.

4.2.3 Communication Reliability

Wireless communications are well known to offer lower reliability guarantees than wired communication links, therefore, attention must be paid to the robustness against likely communication errors. Chapter 3 described null stream values as means to implement region-based computations, as well as to handle possible communication failures. The SOSNA semantics neither specify the requirements nor give any guarantees for the reliability of the underlying communication medium. Instead, the language pushes the reliability issue down to the underlying runtime. The main reason for such a treatment is the fact that, although communication failures are unavoidable, generality considerations lead to the conclusion that addressing the problem beyond the point of its recognition, would unnecessarily complicate the language design simply because reliability levels vary among different communication media and protocols. Nevertheless, since tolerance to message loss is a crucial characteristic of many control algorithms, the question of robustness to communication failures is an important one.

Message loss can affect two general aspects of the operation of SOSNA applications: time synchronisation and program state. In the first case, it is up to the runtime to detect the situation in which individual nodes have desynchronized or that the time source has stopped functioning. The SOSNA middleware uses the FTSP protocol to synchronize time across the network and the protocol can handle such situations by design (Maróti et al. 2004). At the beginning of every application round every network node can state, based on the information from FTSP, whether it is synchronized with the time source. Upon discovery of lack of synchronization, the middleware stops the execution of the SOSNA application and switches into the time synchronization mode awaiting until FTSP regains synchronization. When that happens, the application is reinitialised and restarted.

The loss of application messages, i.e., messages carrying data items aggregated by the application's communication operators, affects the result of subsequent computations in an application-specific way. Although SOSNA requires that the middleware minimises the probability of such situations, the applications should always be aware that 100% communication reliability guarantee cannot be provided. There are many ways applications can handle possible communication failures, i.e. null results of stream aggregation and propagation, and the selection of a particular technique may differ among applications. For example, Section 3.4 describes the use of default and past stream values in recursive stream definitions that involve communication operators. Also, the section shows how one can use stream filtering to implement computations that have some minimal quality requirements (the `avg` function in Listing 3.3).

The prototype SOSNA middleware takes the best-effort approach to communication reliability and does not address the loss of application messages. However, although such an implementation would be insufficient for many real-world applications, alternative solutions are possible. The implementation of reliable communications in WSNs has two important aspects: maintenance of a stable network topology that is based on wireless link quality estimation and the implementation of communication error detection and recovery techniques. There is a body of work addressing the former aspect (e.g., Polastre et al. 2005) and techniques based on message overhearing permit implementations that have low communication overhead. In the latter case, although the real-time requirements add up to the complexity, there is a growing interest in the WSAN research community and promising results have already been achieved (Munir et al. 2010). Also, the industrial-strength WirelessHART protocol already implements reliable real-time communications with IEEE 802.15.4 radios using time-slotted message retransmissions and channel hopping (HART Communication Foundation 2010).

4.2.4 Duty Cycling

The core characteristic of the SOSNA programming language and, hence, of the synchronous macro-programming model is the ability of the compiler generate code for duty cycling network operation. Because the prototype SOSNA compiler targets TinyOS-supported mote platforms, duty cycling motes' microcontrollers is already handled by the operating system. Therefore, in the rest of the chapter we focus on duty cycling the operation of the radio transceiver noting that whenever motes switch their radios off, they could also be explicitely put in a low-power mode if this was necessary. There are four situations in which network nodes can switch their radios off during application execution.

Topology-directed radio duty cycling exploits the fact that multi-hop stream aggregation and propagation is performed by activating nodes at different tree levels, one level at a time. For example, when aggregating stream values along a four-hop deep cluster topology, only the nodes that are supposed to sending and receive data values in a given round should have their radios switched on. Therefore, in the first round of aggregation the nodes at levels one and two can switch their radios off, while in the last, fourth aggregation round, the nodes at the levels three and four can safely switch the radios off. This mode of operation can be used with both static and dynamic cluster topologies because network nodes know at runtime their position within the topology. Also, a similar approach is implemented by Madden et al. (2005) in their work on the TinyDB system.

Stream filtering-based radio duty cycling leverages the property that sensor nodes can establish at runtime whether they need to send or receive messages in a given application round. Firstly, message sending can be omitted when all message's data items take null values. Secondly, the semantics of neighbourhood stream aggregation allow sensor nodes to establish after the evaluation of the round's instant expressions whether this round's aggregation initialisation streams take null values in which case no data items should be aggregated. Network nodes, therefore, are able to assess at runtime both whether a message should be sent and whether any messages should be received. When the answer to both questions is negative, nodes can switch their radios off for the remaining time of the application round. If a message needs to be sent but no messages need to be received, the radio can be switched off after the sending has finished. In all other cases, the radio needs to either remain on, or needs to be switched on of it was previously off.

Heterogeneity-driven radio duty cycling can be used in situations in which different device classes have different communication patterns. For example, in a base station-centric application, the base station usually receives the result of the in-network data transformations at the end of the application cycyle in which case its radio would need to be switched on only in the last application round. This information is available to the compiler at compile time and it is possible to optimise the target code by taking it into consideration.

Time synchronisation duty cycling corresponds to duty cycling radio operation during execution of the time synchronisation protocol, i.e., within service rounds. Spreading of the synchronisation wave is essentially a case of stream data propagation, hence, topology-directed radio duty cycling can be used to minimise energy consumption during this phase of application operation. The SOSNA middleware uses the value of the L_{max} constant as the topology depth and switches the radio on only in those rounds in which the synchronisation messages should be sent and received. The duty cycling mode is used only when a node is time synchronised with the network time source and during the initialisation stage, as well as, when it looses synchronisation the radio is switched on until synchronisation is obtained.

The SOSNA compiler implements the first three duty cycling techniques in a unified way by generating code that dynamically switches the radio off when the message buffer's data item presence bit field indicates that there is no data to send. This is done by generating a code that clears this field at the beginning of every application round and then checks its value after the round's instant expressions have been evaluated. As regards message reception, all network nodes maintain a bounded-size list of data item handlers that are active in the given round. Similarly to the data item presence field, the handler list is emptied at the beginning of every round and then its state is checked once the round's instant computations end. Duty cycling of the time synchronisation protocol, although similar in nature, is implemented by the Sosna Round Protocol and it relies entirely on the current cycle's number, as described in Section 4.2.2. The performance of this radio duty cycling technique is evaluated in Chapter 5 where we analyse the total radio-on duration of applications running on networks of TelosB motes.

Duty cycling the radio transceiver is an operation that is related to network communication reliability because erroneous radio on/off patterns may lead to message loss. The duty cycling policies outlined above make use of the fact that, thanks to the synchronized execution, individual network nodes know their communication patterns and, hence, are able to decide at run time whether it is safe to switch their radios off. Imprecision in network time synchronization or communication delays caused, for example, by the medium access protocol could lead, however, to situations in which messages are not delivered to their recipients. Similarly to the discussion on communication reliability in Section 4.2.3, we require that the underlying runtime provides *timely* communications because the

SOSNA semantics state that messages delayed beyond the duration of one round of data aggregation should be discarded. As regards the imprecision of the time synchronization protocol, tolerance levels can be adjusted by switching the radios slightly before and after the scheduled sleep intervals. Although the prototype SOSNA middleware does not implement this functionality, we analyse the effects of time synchronization imprecision on the synchrony of network operation in Section 5.2.

Chapter 5

Evaluation

This chapter analyses the operation of SOSNA applications compiled to the NesC target and running on a wireless network of TelosB motes. Three generic application scenarios are used in order to demonstrate a range of different operational characteristics of SOSNA applications. The scenarios exemplify three different types of applications: base station-centric, low-power sensor data aggregation, heterogeneous networked actuation and target tracking. The evaluation of the obtained results is divided in two parts. First, we show that the applications execute in conformance with the synchronous macro-programming model, i.e., that their execution is synchronous and that it follows the global communication schedule. To this end we measure the degree of synchronisation of the application cycles among the motes and then we compare the motes' message sending and reception patterns to the expected values, as well as, we present the overall values of the application output, as reported by some of the motes to the PC. Then, we evaluate the WSAN-specific characteristics of application execution by means of the analysis of energy-efficiency, synchronisation of sensing and actuation and binary code sizes of SOSNA applications.

Energy-efficiency is inversely proportional to the rate at which network nodes consume their battery power and we provide accurate measurements of how battery voltage changes in time in a network comprising two types of nodes: those that duty cycle their radios, as well as, those that never switch their radios off. We show that the radio transceiver puts the heaviest strain on mote batteries supporting this way our view that duty cycling radio operation is crucial to the development of truly energy-efficient WSANs. Having established that, we analyse the operation of the three application scenarios in terms of the amount of time the radio was switched on and we show that SOSNA can be used to build sophisticated WSAN applications that make very little use of the radio transceiver.

The results of sensing and actuation synchronisation show that the maximum difference in sensor sampling and actuation times network-wide is in the order of several milliseconds demonstrating this way that SOSNA can be used to build timely and predictable WSAN applications. Finally, we show that the compiler can generate code that fits on the mote-class hardware devices and which makes efficient use of RAM memory which itself can be very limited on some hardware platforms. The results demonstrate that SOSNA, can be used for the development of efficient WSAN applications that can be run on the existing hardware platforms.

The rest of the chapter is organised as follows. Section 5.1 explains the technical background of our experiments, as well as, it describes the three generic application scenarios that we use as the basis of the evaluation. The synchronous execution of SOSNA applications is analysed in Section 5.2. In Section 5.3.1, we show the difference between battery power consumption of motes that duty cycle their radio operation and of motes that never switch their radios off. Furthermore, we analyse the application scenarios in terms of the amount of time the radios were switched on and, in Section 5.3.2, we present the results of sensing and actuation. Finally, we discuss the memory overhead of SOSNA programs in Section 5.3.3 and we conclude the chapter with a summary in Section 5.4.

5.1 The Experimental Setup

All experiments presented in this chapter were performed on networks of TelosB motes. The devices are sold by Crossbow Technology (Crossbow Technology 2009), are powered by two AA batteries and have 10kB of RAM memory, 48kB of program memory and 1MB of flash storage. The motes are driven by the Texas Instruments MSP430 low-power microcontroller, which is a 16-bit processor operating at the frequency of 8MHz and which draws $1.8mA$ current in the active mode and $5.1\mu A$ in the sleep mode. The motes also have an IEEE 802.15.4/ZigBee compliant wireless transceiver capable of data rates of up to 250kbps and they include a light, temperature and humidity sensors. The transceiver consumes $23mA$ in the transmit/receive mode, $21mA$ in the idle mode and $1\mu A$ in the sleep mode. Figure 5.1 presents a TelosB mote network running one of our experiments.

We used the TinyOS 2.1.0 and the NesC 1.2.4 compiler to build the binaries for the TelosB motes. The PC-side support software used the standard TinyOS java support library and was run on the SUN Java 1.6.0_16 virtual machine under the Eclipse Platform version 3.4.2, on the Dell Optiplex 270 desktop that had 1GB or RAM memory and that run Ubuntu 9.04 Linux with kernel version 2.6.30.

Most of the presented results were obtained by periodically logging a number of parameters to the motes' flash storage. Then, after an experiment was finished, a separate application was uploaded

Figure 5.1: TelosB motes during the evaluation of the target tracking application scenario.

to each mote and the logged entries were downloaded to the PC. The logging component was being invoked at the beginning of every application cycle, thus, only when the nodes were time-synchronised and the application was running. Every log entry comprised the contents of the following NesC structure.

```
typedef nx_struct SosnaRoundReportMsg{
    nx_uint32_t global_time;
    nx_uint16_t radio_on_time;
    nx_uint8_t radio_off_count;
    nx_uint8_t received_messages;
    nx_uint8_t processed_messages;
    nx_uint8_t sent_messages;
    nx_uint8_t sensor_use;
    nx_uint16_t voltage;
}
```

The total length of the structure is 13 bytes and we used the NesC network types (the "nx_" prefixes) in order to facilitate the transmission of the logged entries to the PC. The fields correspond respectively to: the global network time at the beginning of the cycle, the total time the radio was on in the previous cycle, the number of times the radio was switched off in the previous cycle, the total number of received messages in the previous cycle (including all overheard application and time synchronisation messages), the total number of *data items* processed in the previous cycle[1], the total

[1]Due to a bug in the logging component that we noticed after having finished all experiments, the received message counter was increased each time a message's data item handler accepted an item instead of each time an application

number of sent messages in the previous cycle, a bit field denoting which sensors were sampled in the previous cycle, and the battery voltage level, as reported by the microcontroller's internal voltage sensor, at the beginning of the cycle.

We used the standard flash logging component `LogStorageC` and the maximum flash volume of 1048576 bytes. The internal voltage sensor reports 16-bit readings that are in the range of $0 - 4095$ units and they can be translated to Volts using to the following formula[2].

$$V = \frac{Reading \times 3}{4095}$$

The transformation results in the voltage levels in the range of $0 - 3V$ and in the case when the initial battery voltage is higher the sensor still reports the value 4095 which corresponds to $3V$. The battery voltage level is directly related to the battery's remaining power and can serve as an indicator of the battery energy consumption. The microcontroller's data sheet (Texas Instruments 2004) states that supply voltage can range from $1.8V - 3.6V$, therefore, when the battery voltage is above $3V$ we consider it full and when it drops below $2V$ we consider it empty.

All of the experiments involved reporting results to the PC. We implemented an actuator driver that was sending a message over the motes' serial connection to a simple Java application that was displaying the contents of this message on the screen and then it was saving them to a file. For that purpose, we used the standard TinyOS components for serial communications (`SerialActiveMessageC`) and the standard TinyOS Java tool chain that handles message reception over the serial link and that allows to automatically generate Java classes responsible for mapping of the message's contents to Java data types (the `mig` tool).

5.1.1 Application Scenarios

We use three different application scenarios to analyse the energy efficiency and timeliness of SOSNA applications. The scenarios realise respectively: multi-hop stream aggregation along a static cluster topology, actuation in a heterogeneous network comprising sensors, actuators and controllers, and the event-based operation of a target tracking sensor network. The reasons behind the choice of these scenarios are threefold. Firstly, they are intended to demonstrate that SOSNA can be used to develop a diverse class of WSAN applications. Tree-based data aggregation is one of the most fundamental applications in which sensor nodes periodically forward their sensor readings towards the tree root (the sink) aggregating them along the way. Synchronised actuation which involves periodic sensor data aggregation at network controllers communicating with each other in order to reach agreement

message was accepted for processing. We include this field for completeness however.

[2]According to the documentation in the `VoltageC` TinyOS component for the TelosB platform.

on the commands that need to be sent to the actuators can be thought generic in such application domains as, for example, building management systems. Tracking a moving target, on the other hand, is a radically different application that requires sensor nodes to actively collaborate in the physical object's proximity and to remain dormant elsewhere.

Secondly, we implement the three scenarios in such way that, instead of processing their sensor readings, the motes compute other parameters such as the degree of synchronisation of sensor sampling and actuation, or application packet loss, so that we can quantify the application's operational characteristics.

Finally, the scenarios demonstrate how SOSNA applications operate under three different models of application dynamics. In each case, we used different values of the three core parameters that define the temporal aspects of the application's operation: the duration of the application cycle, the duration of a single application round and the time synchronisation refresh period L_{max}. In the tree aggregation scenario (TA), the duration of the application cycle is $20s$, hence, the scenario defines a low-rate application that refreshes the time synchronisation every 80 seconds ($L_{max} = 4$) and whose rounds last $500ms$. The distributed actuation scenario (DA) has $8s$ application cycle duration, refreshes the time synchronisation every 32 seconds ($L_{max} = 4$) and has $300ms$ long application rounds defining, therefore, a medium-rate application. The target tracking scenario (TT) defines a high-rate application whose application cycle is $1.4s$ long, time synchronisation is refreshed every $11.2s$ ($L_{max} = 8$) and whose rounds last for $200ms$. The following sections present all three scenarios in more detail.

Tree Aggregation

The tree aggregation application scenario exemplifies multi-hop stream aggregation along a static cluster topology. The SOSNA program implementing the scenario is presented in Listing 5.1. The program computes the maximum difference in light sensor sampling times, as reported by the **`time`** operator (line 10), by aggregating a stream of tuples comprising the earliest and the latest sensor sampling times throughout the entire network (lines 12-15). The nodes use their current sampling time as both the minimal and the maximal sampling times (line 14) and `maxSpread` function (lines 5-8) separates the local minima from the maxima calculating the final result at the base station mote using a maximum time spread as the initial value (line 13). Additionally, the application calculates the total number of data items contributing to the aggregation of the sensing times by adding up the value 1 for every node (line 17). Knowing that both aggregation expressions are independent and that the compiler schedules them for execution in parallel, there is no need to explicitly merge them, as it

Listing 5.1: The SOSNA program for the tree aggregation application scenario.

```
1  (def-class base)
2  (def-class motes : base)
3  (def-input TelosLight @ motes : int)
4
5  (def-fun (maxSpread (tuple newMin newMax) (tuple totalMin totalMax))
6    (let (min (if (< newMin totalMin) newMin totalMin))
7      (let (max (if (> newMax totalMax) newMax totalMax)))
8        (tuple min max)))
9
10 (def sampleTime (time TelosLight))
11
12 (def minS maxS
13   (aggr maxSpread (tuple MAX_TIME (gtime 0))
14                   (tuple sampleTime sampleTime)
15                   (clstat motes)))
16
17 (def nPackets (aggr + (gtime 0) (gtime 1) (clstat motes)))
18
19 (output (Report (tuple minS maxS nPackets)))
```

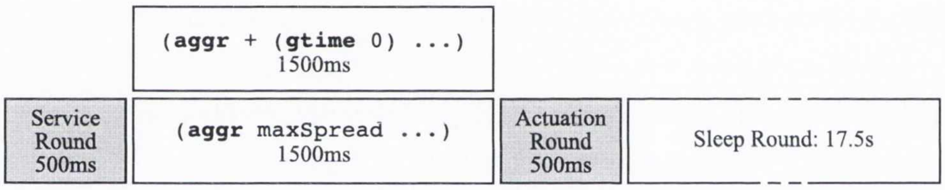


Figure 5.2: Scheduling of the tree aggregation scenario.

was done for the more general case in Section 3.4. The base station, after computing the final result sends it to the PC via the serial connection (the `Report` actuator, line 19). The **`gtime`** keyword in line 17 casts the type of integer constants to the Time data type that represents the application global time and we use it in this context only so that the implementation of the `Report` actuator could be simplified by considering only tuples of elements of the same type. This limitation, however, bears no significant technical reason and can be removed in the future versions of the compiler.

We used the following compilation parameters for the evaluation of the scenario. The maximum static cluster depth (the number of application rounds comprising an aggregation operation over a static cluster) was limited to three hops. The duration of a single application round was set to $500ms$ while the sleep round was set to last for $17.5s$ resulting in the application cycle of 20 seconds (see Figure 5.2). The time synchronisation period was set to four application cycles resulting in the refresh rate of 80 seconds. The experiment was running for approximately twelve hours.

The scenario was deployed on a network of sixteen TelosB motes in an office environment. A

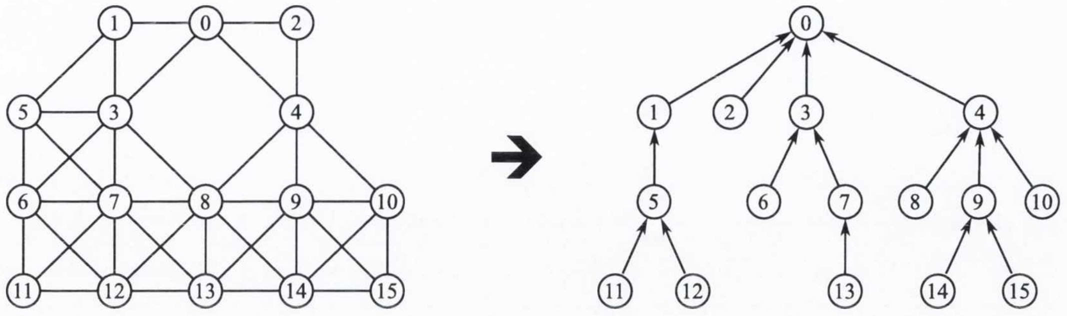


Figure 5.3: The logical network topology of the TA application scenario and an example static cluster topology built over the logical topology.

logical communication topology was imposed on the network in order to allow the SOSNA middleware to construct a three hop-deep cluster topology along which the application aggregated its data (see Figure 5.3). The logical topology was defined by means of network identifiers that were assigned to the motes at the deployment time.

Distributed Actuation

The distributed actuation scenario demonstrates a case for synchronised actuation over a multi-hop heterogeneous network, similar to the distributed data aggregation and actuation presented in Section 3.4 (Listing 3.1). The scenario is intended as an example of a *generic* distributed control application that comprises three different classes of network nodes: controllers, sensor nodes and actuator nodes, communicating over static cluster and neighbourhood topologies in order to implement an abstract control law. We divided all network nodes in two groups that were deployed in two neighbouring offices distinguished by their numbers - 214 and 215. Each of the rooms had its own set of sensors and actuators, and there were two controllers of which one was located in the office 215 while the other was placed in the corridor connecting both rooms.

The application's goal was to collect sensor readings from both rooms independently, then to use these values to compute control commands for the actuators in each of the offices separately. The information was flowing from the sensor nodes to the controllers which were synchronising their state in order to compute the control commands which were then sent to the actuator nodes for execution. Although we used the motes' light sensors in order to generate the sensor readings, we did not intend to utilise in any meaningful manner. Instead, we use the scenario, in addition to demonstrating a complex heterogeneous WSAN application, to quantify the degree of actuator synchronisation in SOSNA applications. The actuator nodes, therefore, apart from calling their actuator drivers were also aggregating the actuation times recorded in the previous instant, in the same way as sensor sampling times were aggregated in the TA scenario. The final aggregation results were delivered to the controllers which again synchronised the received values and reported them to the PC.

Listing 5.2 presents the SOSNA application implementing the scenario. Lines 1-3 define the three device classes that correspond respectively to the controller nodes, the sensor nodes and the actuator nodes. In order to measure the actuation times, we use a virtual actuator `SaveActuation` (line 46) that, instead of controlling a hardware component, simply measures the global time at which it was invoked. The measured value is read in the following instant by a virtual sensor driver `LastActuation` (lines 5, 42) and aggregated in much the same way as in the TA scenario (lines 35-44). The only difference is that we extend the aggregated tuples with an additional element (`total_n`) counting the number of nodes that contribute to this aggregation process.

The program aggregates the light sensor readings and the actuation times towards both controllers which share these values among each other so that the control commands can be computed. We implement the aggregation and sharing processes as a single stream function `shared_aggr` (lines 8-10) that takes five arguments representing respectively the aggregation function, the initial aggregation value, the aggregated stream, the destination device class (`class_a`) and the source device class (`class_b`). The function simply composes two stream aggregation operations: one along the static cluster topology of the source device class and the other one along a one-hop neighbourhood topology of the destination device class (the data sharing part). We use the function to find the maximum spread in actuation times throughout the network (lines 35-44) and to calculate average light levels in each of the offices.

The `av_reading` function (lines 12-18) implements a generic spatial stream averaging procedure that operates in a similar way to the one in Listing 3.1 but with the difference that it uses the `shared_aggr` stream function instead of the plain **`aggr`** operator. The function is used twice - each time for a different room (lines 24,30) thus calculating average light levels for each room separately. The average values are used to compute control commands for the actuators (lines 23,29) that we define in a simple way as a difference between the desired and the current light levels (line 20). The control commands are then forwarded to the actuators (lines 25-27 and 31-33) and used to exercise actuation in line 46. The final result of the application's operation constitute the values of the maximum spread in actuation times, the number of actuators that contributed to the computation of the aggregate actuation time spread (so that we can quantify packet loss) and the average light levels in both rooms. We pack these values into a single tuple of integer values and send them to the PC (lines 47-48).

Similarly to the TA scenario, we used a logical communication topology so that all stream aggregation and propagation operations could be performed over a two-hop cluster topology. We run the

Listing 5.2: The SOSNA program for the distributed actuation application scenario.

```
(def-class base)
(def-class sensor : base)
(def-class actuator : base)
(def-input TelosLight @ sensor : int)
(def-input LastActuation @ actuator : int)
(def-extern ROOM : int)

(def (shared_aggr f init stream class_a class_b)
  (let (result (aggr f init stream (clstat class_b)))
    (aggr f result result (nhood class_a))))

(def (avg_reading region sensor class_a class_b)
  (let (f (lambda ((tuple new_s new_n) (tuple sum_s sum_n))
           (tuple (+ new_s sum_s) (+ new_n sum_n))))
    (let (s (within region (tuple (vals sensor) 1)))
      (let ((tuple sum_s sum_n)
           (shared_aggr f (tuple 0 0) s class_a class_b))
        (/ sum_s sum_n)))

(def-fun (control_law avg_reading) (- 500 avg_reading))
(def-fun (forward val init) val)

(def cmd_215 (smap control_law
                (avg_reading (== ROOM 215) TelosLight base sensor)))
(def act_215 (expand forward (within (== ROOM 215) 0)
                cmd_215
                (clstat actuator)))

(def cmd_214 (smap control_law
                (avg_reading (== ROOM 214) TelosLight base sensor)))
(def act_214 (expand forward (within (== ROOM 214) 0)
                cmd_214
                (clstat actuator)))

(def-fun (maxSpread (tuple new_min new_max new_n)
                    (tuple total_min total_max total_n))
  (let (min (if (< new_min total_min) new_min total_min))
    (let (max (if (> new_max total_max) new_max total_max))
      (tuple min max (+ new_n total_n)))))

(def min_s max_s act_n
  (let (t (time LastActuation))
    (shared_aggr maxSpread (tuple MAX_TIME (gtime 0) 0)
                           (tuple t t 1) base actuator)))

(output (SaveActuation (merge act_214 act_215))
        (Report (tuple (gtime->int (- max_s min_s))
                       act_n cmd_214 cmd_215)))
```

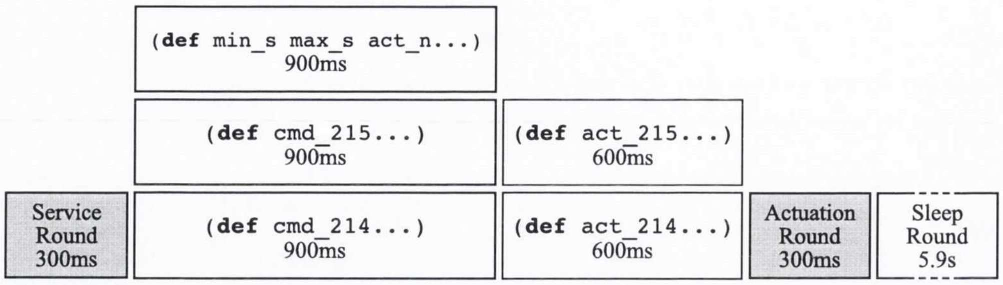


Figure 5.4: Scheduling of the distributed actuation scenario.

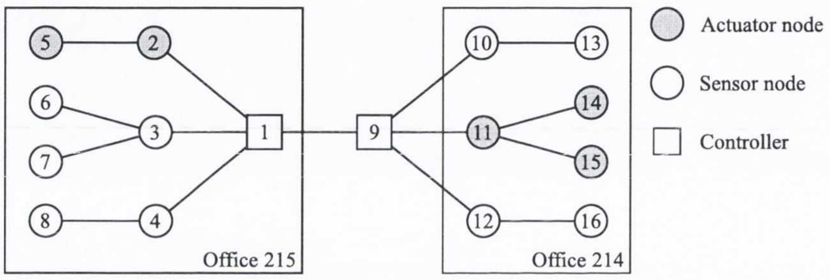


Figure 5.5: Experimental setup of the distributed actuation scenario.

application on the network of sixteen TelosB motes and Figure 5.5 presents the overall experimental setup. We used the following compilation parameters. The application round duration was set to $300ms$ and the sleep round was set to $5.9s$ so that, given the maximum static cluster depth of two hops (see Figure 5.4), the duration of the application cycle was $8s$. Network time synchronisation was refreshed every four application cycles, hence every 32 seconds. The experiment was running for approximately thirteen hours.

Target Tracking

The target tracking application scenario demonstrates how event-driven WSAN applications can be developed in SOSNA. The previous two application scenarios were based on communications performed over static cluster topologies, hence, their operation was regular, i.e., repeatable in every instant, as well as, it involved all network nodes. Event-driven applications involve activities that are performed only when an event of interest occurs and usually by nodes in relative proximity to the event detection area.

Target tracking is a canonical application in which the position of a physical object is being collaboratively estimated and tracked by the network nodes, as the object moves within the network's coverage area. We evaluate the performance of a target tracking application that is similar to that presented in Section 3.4 (Listing 3.2) but which differs in one important respect. Instead of using real sensor readings, we define the object's position computationally, i.e., by making all nodes calculate it based solely on the current value of network time. The reasons for taking this approach are twofold. Firstly, we want to be able to analyse the application's characteristics in a controlled way, thus avoiding the irregularities and non-repeatability related to the detection and moving of a physical object. Secondly, because the main goal of the scenario is to demonstrate how region-based stream operations work in practise, as well as, to show how they can be used to minimise network energy consumption, the choice of the object detection method becomes of secondary importance and by using a simplified method we can rule out all factors that are not directly relevant to the experiment's goals.

The application implementing the target tracking scenario is presented in Listing 5.3 and it operates as follows. First the current instant's number is computed by dividing the network time by the duration of one application cycle (lines 14-16). We use the light sensor sampling time as the current network time because the sensor is sampled at the beginning of each cycle and we additionally increase the value by $10ms$ in order to ensure that all nodes compute exactly the same value[3]. We expect the network nodes to be arranged in a linear topology and to be assigned positions that are sequential numbers in $[0, 15]$. Thus, we can obtain the object's position by dividing the current instant's number modulo 16 so that the object periodically visits all nodes one after another (line 18).

Once the object's position is known, the distance to the object is calculated as the difference between the node's position and the current object's position (line 20), and the object proximity region is defined as the set of all nodes that are not further than two units (hops) away from the object (lines 22-23). Then, those nodes that are in the object's proximity elect among themselves the one that happens to be the closest to the object and that node becomes the temporal tracking leader. We make the leader count the number of proximity group members by aggregating the value 1 for each of them (lines 25-26), in order to reflect the situation in which the leader needs to estimate the object's position based on the tracking group members' partial estimations. Finally, the leader transmits the aggregated value to the base station so that it could be transmitted to the PC (lines 28-30).

The application additionally demonstrates how actuator coordination can be achieved in SOSNA: all nodes that belong to the tracking group actuate by their switching on their LEDs based on their current distance to the object (line 31). Because, once switched on the LEDs continue to emit light, we need to explicitly make all other nodes switch their LEDs off.

We evaluated the target tracking scenario on a network of sixteen TelosB motes and one MicaZ mote

[3]This is a precaution measure and a shortcoming of the existing prototype implementation.

Listing 5.3: The SOSNA program for the target tracking application scenario.

```
(def-class base)
(def-class motes : base)
(def-input TelosLight @ motes : int)
(def-extern POS @ motes : int)
(def N_MOTES 16)

(def-fun (abs x) (if (< x 0) (- 0 x) x))

(def-fun (indicate distance)
  (if (== distance 0) 1
    (if (== distance 1) 2
      (if (== distance 2) 4 0))))

(def cycle_n (gtime->int (/ (+ (time TelosLight)
                               (gtime 10))
                            (gtime 1400))))

(def obj_pos (mod cycle_n N_MOTES))

(def obj_dist (smap abs (- mote_pos obj_pos)))

(def detection (within (< obj_dist 3)
                       obj_dist))

(def obj_estimate (let (leader (clmin detection))
                         (aggr + 1 1 leader)))

(output (Report (aggr (lambda (x y) x) (tuple 0 0)
                        (tuple mote_pos obj_estimate)
                        (clstat motes)))
        (Display (merge 0 (smap indicate detection))))
```

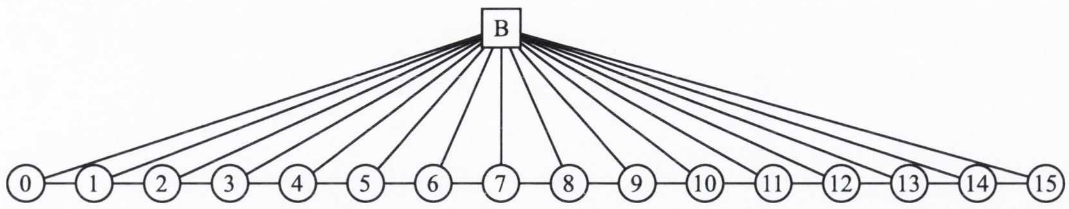


Figure 5.6: Experimental setup of the target tracking scenario.

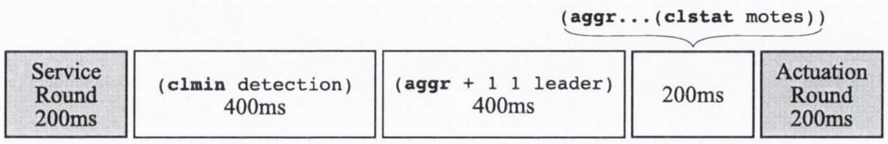


Figure 5.7: Scheduling of the target tracking scenario.

that played the role of the base station. All TelosB motes were arranged in a linear communication topology but each of them was able to communicate directly with the base station (see Figure 5.6). We used the following compilation parameters. The application round duration was $200ms$, the duration of the sleep round was zero while the maximum static cluster depth was set to one hop and the maximum dynamic cluster depth was two hops. As a result, the application cycle duration lasted $1.4s$ (Figure 5.7 describes the application's scheduling). Because the application cycle was relatively short we set the time synchronisation period to be eight cycles, resulting in the $11.2s$ refresh rate. The experiment was running for approximately twelve hours.

5.2 Synchronous Execution

This section investigates whether the execution of the three application scenarios conforms to the synchronous macro-programming model. The most fundamental property of the model is that it imposes a global communication schedule on the operation of the whole network. As it was discussed in Chapter 4, the prototype SOSNA compiler schedules application cycles as sequences of rounds. Each sequence starts with with a service round that is reserved for such tasks like network time synchronisation or sensor sampling. The service round is followed by a sequence of application rounds. Network nodes can send at most one message in every application round but may receive an unspecified number of messages. The sequence of application rounds ends with the actuation round that is a time slot reserved only for computation and actuation. Finally, the application cycles end with the sleep round that is a time period of arbitrary duration, including the duration of zero seconds.

We quantify the degree of synchronised application execution through the measurements of the

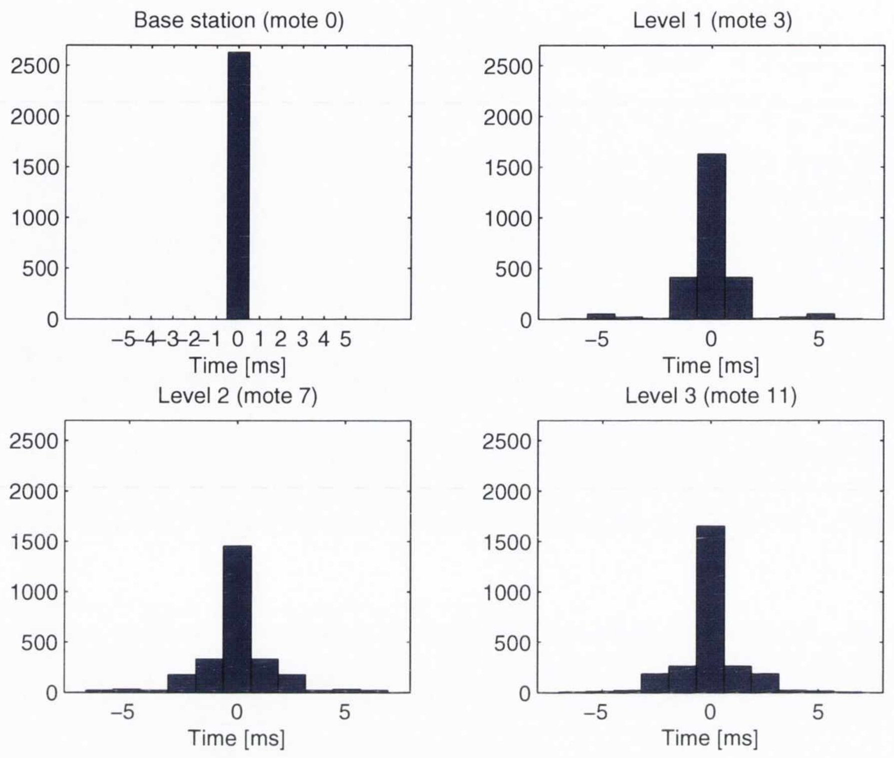


Figure 5.8: Distribution of the cycle synchronisation error in the TA scenario.

global times at which application cycles were commencing at all motes. The global time values were computed by each of the motes through the conversion of the mote's local time to the global time. The conversion was performed by the FTSP component and it involved clock drift compensation which was estimated on the basis of the received time synchronisation messages. The global time readings, therefore, do not represent the physical real time but rather correspond to the current value of the time synchronisation root's local clock increased by an additional synchronisation error which, according to Maróti et al. (2004) should remain within several microseconds. However, since the FTSP implementation that is the basis of our prototype uses timers with binary millisecond resolution (1024 ticks per second), we are limited in the accuracy of our measurements to the unit of little less than $1ms$ (for simplicity the unit is referred to as one millisecond).

We denote the global starting time of the n-th application cycle by T_n and the total duration of the cycle by D_{cycle}. We define the cycle synchronisation error according to the following formula:

$$E_{sync} = T_{n+1} - T_n - D_{cycle}$$

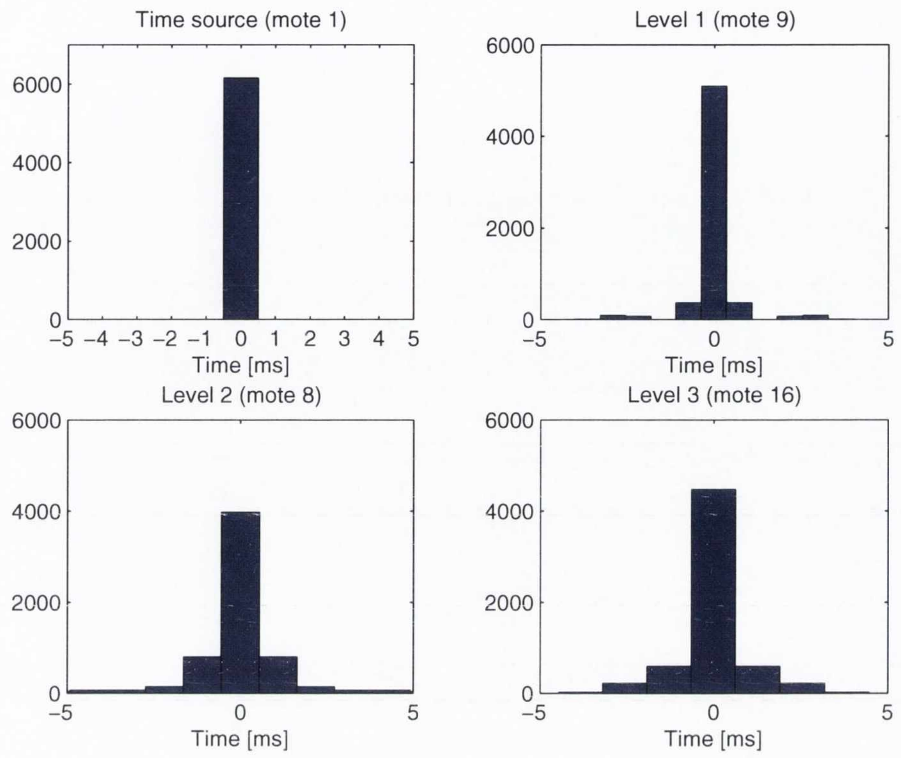


Figure 5.9: Distribution of the cycle synchronisation error in the DA scenario.

When all motes are perfectly synchronised the cycle synchronisation error should be equal to zero meaning that all cycles have the same duration that is equal to the expected value. However, due to the global-to-local time conversion it is possible that the motes miscalculate the starting time of the next application cycle and, when the cycle is about to start, the global time value might be slightly different than previously expected.

Figure 5.8 presents the distributions of the cycle synchronisation error for the data collected during the execution of the TA scenario. The results for all motes follow a pattern in which the cycle synchronisation errors are very similar for the motes located at the same level of the time synchronisation tree. This is why the figure presents the results for only four motes that represent their corresponding time synchronisation tree levels. The results show that with high probability the cycle synchronisation error is bounded by $5ms$, as well as, that the error increases with distance from the time source node. The error values for for the time source mote (the base station) seem to imply that the cycle synchronisation error is dominated by the effects of the operation of the time synchronisation protocol.

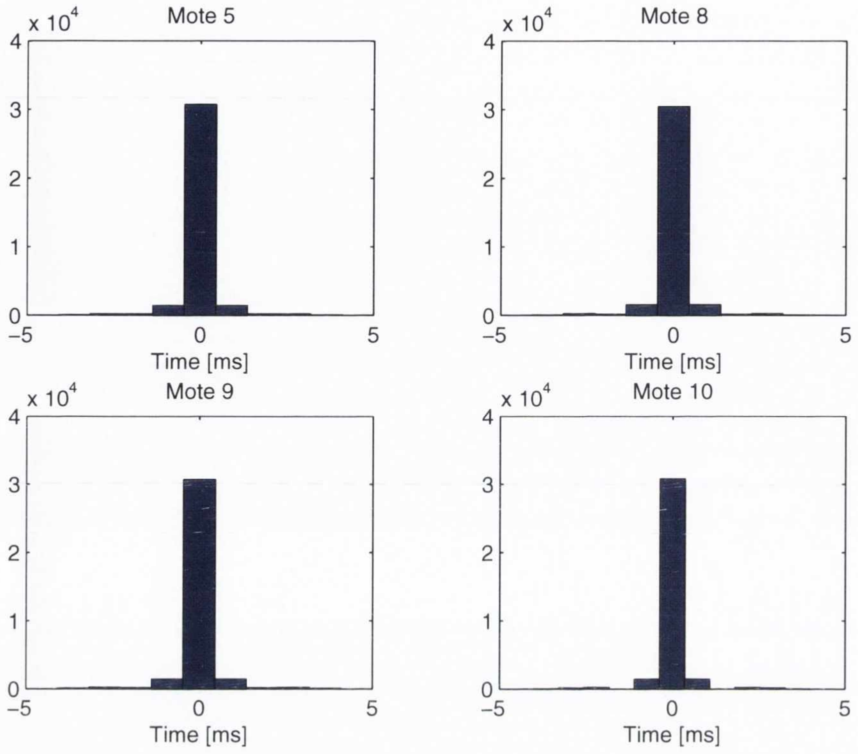


Figure 5.10: Distribution of cycle synchronisation error in the TT scenario.

A similar regularity was discovered for the data coming from the DA scenario and the results presented in Figure 5.9 resemble those from Figure 5.8. The TT scenario arranged all motes in a one hop time-deep synchronisation tree, therefore it could be expected that the cycle synchronisation error values should liken those of the level-1 motes in the TA and DA scenarios, as well as, that the error levels for different motes should be similar. Indeed, this is the case and Figure 5.10 presents the results for the four arbitrarily chosen motes.

Although the synchronisation error in the order of several milliseconds might be sufficient for many WSAN applications, it is important to ensure that it can be tolerated within the synchronous macro-programming model. The synchronous macro-programming model does not specify the required degree of network time synchronisation, nor it suggests the use of a particular time synchronisation protocol. Therefore, it is up to the model implementation to ensure execution correctness.

We did not note any irregularities in the execution either of the three application scenarios or of other test applications that are not discussed in this thesis. The main reason for this is that the synchronisation error is small relative to the duration of the application round (1% for the TA scenario,

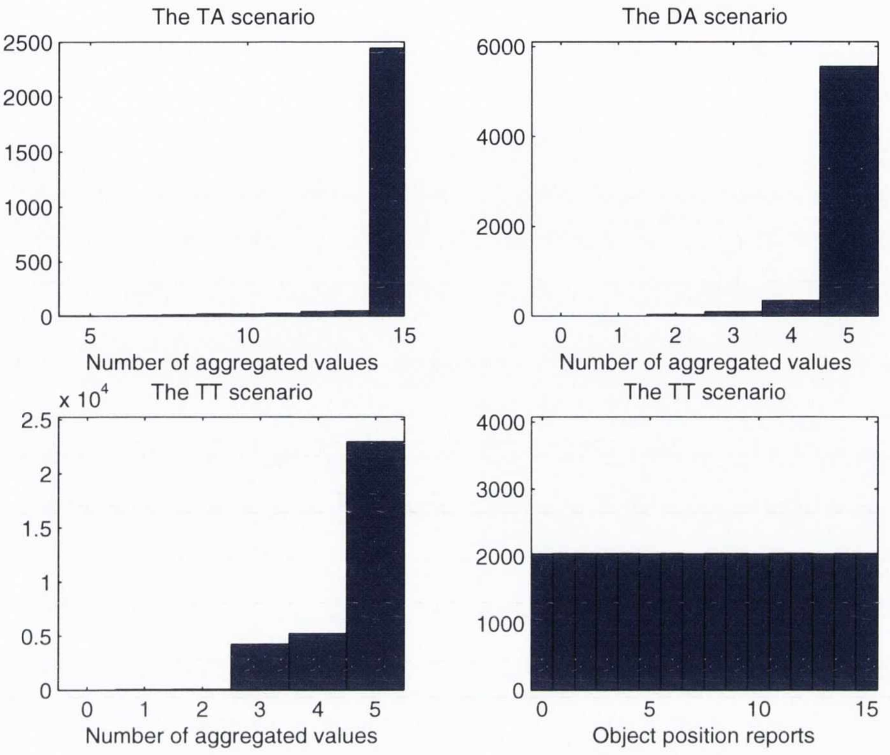


Figure 5.11: Distribution of the number of data items contributing to aggregation in all three scenarios and the distribution of the object position reports in the TT scenario.

1.7% for the DA scenario and 2% for the TT scenario). We review, therefore, the possible outcomes of the situation in which a mote starts its cycle little earlier or little later than expected.

The shift in the cycle starting time affects the starting times of this cycle's application rounds because they are scheduled relative to it. Shifting the starting time of the service round affects the sensor sampling times, hence, it has application-specific consequences. Similarly, a shift in the starting time of the actuation round affects only the time of invoking the actuator drivers, while the shift in the sleep round time has no significant consequences. We focus the analysis on shifting the starting time of the application rounds.

Each application round starts with computation of the round's instant expressions which might be followed by message sending and reception. When an application round is started little earlier, there are two possible consequences. The mote might send its message before the message's recipients switch their radios on or even before the previous round ends resulting in its neighbours perceiving it as out of order and consequently dropping it. This situation, however, is not common because all message sending tasks are preceded by a random delay of one third of the round duration. On the other hand, the rare case when a message is lost, has only local temporal consequences as the model handles message loss and applications recover their state in the next cycle. Also, the computation at the beginning of the round may well cancel the effect of the earlier starting time in its entirety. Starting the round few milliseconds later may only result in delayed activation of the radio for message reception and sending. This, again, is not a serious issue because the mote's neighbours delay their sending times, as well as, that a lost message is handled by the model by definition.

The degree of robustness to the imperfections of time synchronisation can be quantified when we analyse the output values reported by the base stations in each of the three scenarios. For example, the TA scenario realises data aggregation along a static cluster topology that results in the computation of the maximum spread in sensor sampling times and of the number of data values (or data items) that comprised the aggregate value. We defer the discussion of the sensor sampling time spread to Section 5.3.2 and discuss in this section only the number of the contributing data values.

The network comprises fifteen motes (plus the base station), hence the expected result should always be equal to fifteen. Channel collisions and possible time synchronisation errors might lead to a different results being returned. The results for all three scenarios (DA and TT computed analogous values) are presented in Figure 5.11. We can see that in all three scenarios the network was computing correct results and in most cases the computations involved all nodes that were supposed to contribute. We can also see that the number of lost data values slightly increases for the DA scenario and then further in the case of the TT scenario. The network remained operational in all cases, hence the missing values affect only the quality of the reported results (the figure shows that the target tracking application was constantly and correctly tracking the object because all of its periodic locations have been reported the same number of times).

It is difficult to definitively assign a cause to the correlation of increased packed loss with the decreasing cycle duration because each of the experiments was performed in different conditions and our experimental data does not provide sufficient information. Possible contributing factors include random message loss, time synchronisation errors or the experimental setup (all motes in the TT scenario were aligned close to each other, as in Figure 5.1 while in the other scenarios the motes were spread over the entire office area). Whichever the real cause was, there would be ways of tackling it because the current SOSNA execution model is by no means optimal. For example, if the time synchronisation was the source of the problem additional delays could be introduced to the global schedule in order to accommodate the error.

5.3 WSAN-Specific Characteristics

In this section, we evaluate the WSAN-specific characteristics of Sosna applications running in mote networks. We analyse the performance of the Sosna's radio duty policies showing that the language can be used to develop energy-efficient WSAN applications. Then, we briefly discuss the degree of synchronisation of sensor sampling and actuation. These two metrics, although related to the main theme of the previous section, have application-specific semantics, i.e., the acceptable error levels are defined by the application domain rather than being related to the fundamental correctness of our approach. We conclude the section with the discussion of the binary code sizes resulting from compilation of Sosna programs to NesC and then further to the target platform executable code. We show that Sosna applications have low RAM memory usage and that the language can be used to develop applications for highly-resource constrained hardware platforms.

5.3.1 Energy Efficiency

We quantify the energy efficiency of Sosna applications in two ways. First, we show that duty cycling radio operation makes a significant difference in terms of battery energy consumption. This result is in agreement with the results of Pottie & Kaiser (2000) and Raghunathan et al. (2002), as well as, it is widely acknowledged fact within the WSN community. Then, the obtained results are related to the theoretically achievable performance of the BoX-MAC protocol, which is the default low-power asynchronous MAC protocol in TinyOS 2.1.0. Additinally, we discuss the similarities between the Sosna's scheduled tree data aggregation and the low-power data collection protocol Dozer. Finally, we analyse the radio duty cycling schemes of the three application scenarios described in the previous section. We show that in all three cases our approach allows to minimise the use of the radio transceiver without affecting the application's operation.

We use a basic tree aggregation application presented in Listing 5.4 to compare the energy consumption of motes duty cycling their radios with those that never switch their radios off. The application simply aggregates light sensor readings along the static tree topology. The experiment involved five TelosB motes arranged in a two hop-wide communication topology that periodically logged their battery voltage levels. The application round duration was $150ms$ and the sleep round was $5.6s$ long resulting in the application cycle of six seconds. The time synchronisation refresh period was two cycles. The experiment was running for 120 hours (five days) and all motes were using brand new Duracell LR6/MN1500 AA batteries. Two of the motes did not duty cycle their radios while the remaining three did. Figure 5.12 presents the experimental setup and the results. The energy cost of

Listing 5.4: Basic two-hop stream aggregation application.

```
(def-class base)
(def-class motes : base)
(def-input TelosLight @ motes : int)

(def result (aggr + 0 (sync TelosLight) (clstat motes)))
(output (Report result))
```

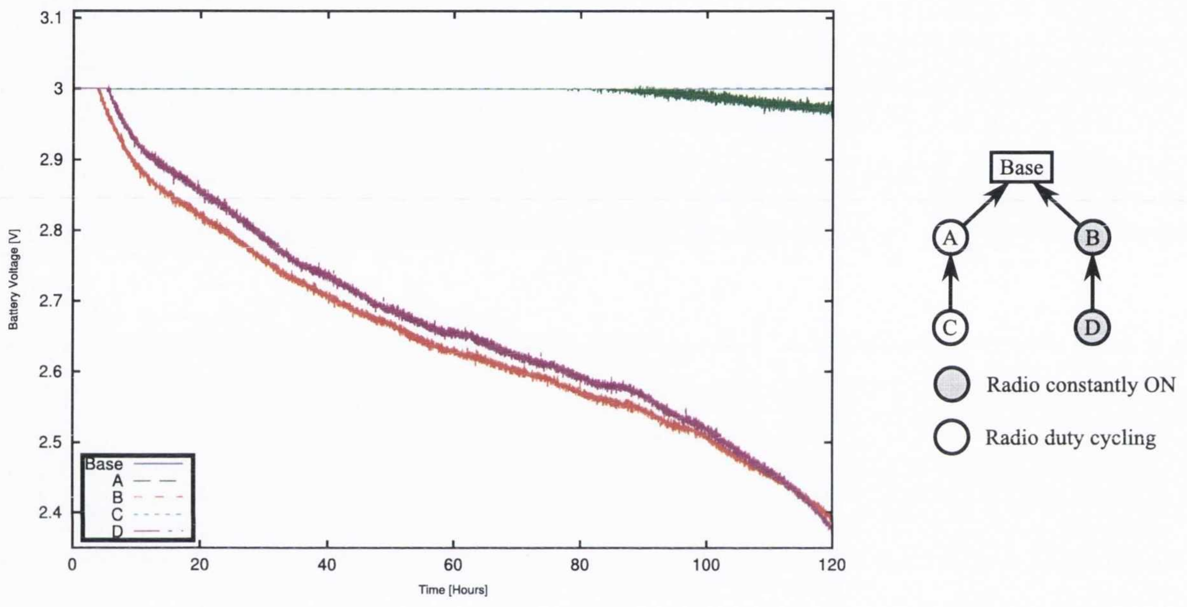


Figure 5.12: Battery power consumption with and without radio duty cycling.

flash logging was not accounted for because it was the same for every mote.

The results indicate that radio duty cycling radio leads to significant energy savings. After five days of operation, the B and D motes exhausted most of their battery power, while the remaining three motes could have continued their operation for much longer time. The flat line at the top of the graph corresponds clipped voltage levels. The clipping was the effect of the voltage sensor's operation for voltage values above $3V$, hence, throughout the experiment's duration the base station and the C motes retained their battery voltage level above $3V$ while the A mote was using up its energy at a higher rate. The difference in power consumption between A and C was caused by the fact that A had to listen for data items from C, thus had to keep its radio on for a longer time. We explain this fact in more detail in the next section. The small difference in power consumption between the B and D motes was most likely caused by a small variation in battery characteristics because both motes sent the same number of messages and did not switch their radios off.

The base station mote and the C mote reported the most energy-efficient operation because they kept the use of their radio transceivers at the lowest level. Every second cycle, the motes were switching their radios on for the duration of the service round ($150ms$) in order to run the time synchronisation protocol. In every application cycle, the C mote was switching its radio on for the time needed to send its data item, i.e., for the duration of the random delay of maximum $150ms/3 = 50ms$ plus the sending time that might involve additional random delay related to channel contention (we assume a maximum of $30ms$). At the same time, the base station node was switching its radio on for the duration of one application round to receive data items from the A and B motes. As a result, when we consider the overall duration of the application cycle, we can say that the C mote was operating at the average duty cycle of 0.026 because:

$$\frac{150ms + 2 \times (50ms + 30ms)}{6000ms} \times \frac{1}{2} \approx 0.026$$

Similar calculations lead to the result that the base station mote and the A mote were operating at the duty cycles of 0.036 and 0.05 respectively. We use these values to give a sense of the battery power consumption in the three application scenarios presented in the following sections.

As regards the effect of duty cycling on message lost, the experiment showed no significant difference among the duty cycling nodes and those that kept their radios on at all times. The A, B and Base motes were expected to receive two messages in every instant because, due to the physical proximity, all motes were in each others communication range. The percentage of messages received by the motes were 88.838%, 88.84% and 98.967% respectively. The noticeable difference in the message loss ratios among the Base node and the A and B nodes, although not related to the effect of duty cycling, requires further investigation because the existing experimental data does not permit to draw definite conclusions. However, it is likely that environmental interference can be put among the significant factors since the experiment was performed in a small apartment located just above a busy intersection.

Comparison to BoX-MAC

The obtained results clearly demonstrate the importance of duty cycling for battery-powered devices. They also show that radio transceivers consume a significant amount of energy while in the idle listening state. However, leaving the radios on at all times is a strategy only few would decide to use in more realistic scenarios. Instead, programmers have to either develop their own duty cycling communication protocols or resort to a generic technique called low-power listening (LPL), originally proposed by Polastre et al. (2004). The technique is based on the idea that receivers periodically switch their radios on to check whether there is a transmission going on, while senders precede their transmissions with preambles that are sufficiently long for the receivers to pick them up. Therefore, at the increased cost of packet transmission, motes save energy on idle listening.

TinyOS 2.1.0 implements a low-power listening medium access protocol called BoX-MAC (Moss & Levis 2008). The protocol repeatedly transmits the first packet sent after the radio was woken up (the preamble) while awaiting at the same time an acknowledgement from the receiver. The receiver, on the other hand, periodically switches the radio on in order to measure channel energy levels (clear channel assessment - CCA). Once channel activity has been detected, the receiver is able to receive the following packet retransmission. After the packet has been acknowledged, both the sender and the receiver retain their radios on for a short while allowing this way for subsequent transmissions to take place without the need to transmit the preamble again. The main advantages of this scheme are, thus, the ability to shorten the preamble duration thanks to reception acknowledgements, as well as to increase data throughput once the radios have been switched on.

Although, BoX-MAC achieves good results with point-to-point transmissions, broadcast communications pose a problem since it is costly to implement reception acknowledgements in the presence of multiple receivers. This makes the use of the protocol tricky in application scenarios involving neighbourhood data sharing. In particular, the the protocol would be of little use for SOSNA because the compiler packs multiple data items into a single message and, since the messages often have multiple destinations, broadcast transmissions are used. Nevertheless, in order to give a better sense of the energy savings the protocol can achieve, we can estimate its energy consumption for the above example application.

In order to make the comparison more meaningful, we analyse the operation of an application that is *functionally equivalent* to the above tree aggregation scenario. We consider, therefore, an application that samples sensors once every six seconds, transmits their values towards the base station while aggregating them along the way, and that synchronizes network time in order to synchronize sensor sampling. We do not require, however, that the messages be sent at any particular times, as it is in the case of the SOSNA's communication schedule. The application has the same communication patterns as the one in Listing 5.4, hence the A and B motes are the busiest ones: in every period each of the motes has to send one message to the base station, receive one message from the C or D mote, receive a time synchronization message from Base or propagate it down to C or D. In order to estimate the average radio on time we use the formula derived by Moss & Levis (2008) using the duration of two periods ($12s$) instead of one day. The following formula describes a rough estimation of the *average* time the radios of A and B motes are on in every two consecutive execution periods :

$$r_{on}(T) = \frac{P}{T} \cdot 5.61ms + V \cdot 50ms + I \cdot 20ms + D \cdot \frac{T}{2}$$

The meaning of the parameters is the following: T is the period between receive checks in milliseconds, P is the duration of the measurement period, V is the number of valid packet receptions, I is the number of invalid packet receptions (destined for a different node), and D is the number of transmitted packets. By taking $P = 12000ms$, $V = 3$, $D = 3$, $I = 7$ (all overheard messages, assuming that all nodes could hear each other, as in the original experiment), we get:

$$r_{on}(T) = \frac{67320}{T} + T + 290$$

Since the optimal choice for T that minimises r_{on} is $T = 164ms$ and $r_{on}(164ms) = 925.55ms$, we find that the radios of the A and B motes would be active 7.71% of time. Although this is a fairly good result, we can see that SOSNA's communication scheduling gives way to duty cycles that are two times lower. Also, even better results could be achieved if SOSNA switched the motes' radios off immediately after they have transmitted the time synchronization message (currently the radios stay on for the duration of the whole service round and, hence, the value of $150ms$ was used in the calculations). Duty cycles could be lowered even further if the application's sleep period was extended, in which case, BoX-MAC would consume more energy due to the continuous need to assess channel activity. Finally, real experiments would have to be performed in order to state BoX-MAC's real energy cost since the formula does not account for the complexities of channel contention and random back off.

Comparison to Dozer

Dozer (Burri et al. 2007) is a data collection protocol that uses lightweight TDMA schedules in order to synchronize communications in the network so that the nodes can switch their radios on only when they need to transmit data. The authors observe that the traditional TDMA MAC protocols are difficult to implement on resource-limited wireless networks due to the cost of establishing and maintenance of a global communication schedule. Dozer tackles this problem by exploiting the fact that in many typical WSNs data is collected along a tree network topology towards a single base station node. Thus, instead of a global universal communication schedule, the protocol schedules communications locally among tree parent nodes and their tree children. The protocol does not require global time synchronisation and it uses beacon messages to synchronise every parent with its children for the duration of the subsequent communication period. The structure of the local communication schedule reflects the expected direction of communication flow in the network: in every communication period data is sent by tree children to their corresponding tree parents.

The protocol implements the global tree topology in a robust and reliable way. The children nodes select their parents based on communication link quality and the parent's load (i.e., the number of its children), thus balancing the topology structure. Packet acknowledgements are used for all data travelling up the tree and the protocol adjusts to prolonged link failures and topology disconnections.

Dozer uses beaconing to maintain the communication schedules throughout time. The beacons are radio messages that inform the parent node's children about the current communication schedule and the starting times of the next schedule, as well as they can be used to send short commands down the data collection tree. The protocol randomises starting times of the subsequent communication periods in order to avoid interference with those network neighbours that are members of a different schedule (children of a different parent node). This information is embedded in each beacon message and it causes the duration of each communication period to float around a mean value. Thanks to its flexible communication scheduling, Dozer is capable of very low-power operation. The authors report that, in an experiment with 40 motes deployed in an office building, the network was running with an average duty cycle below 0.2%.

Although SOSNA is a programming language that can implement much more sophisticated communication schemes than tree-based data aggregation, it is interesting to compare network energy consumption of the example data aggregation program from Listing 5.4 to that of Dozer. Burri et al. (2007) report that the 0.2% duty cycle was achieved with the communication period duration of two minutes. The authors also relate duty cycles to the number of node's children and to the duration of the communication period. As expected, the duty cycle increases with the number of children grows and it decreases as the communication period gets longer. Our experiment with the example SOSNA tree aggregation application resulted in duty cycles of 3.6% in the case of the A mote, 5% for the base mote and 2.6% in the case of the C mote (the communication period was $6s$ long). These values could be significantly lowered if a longer sleep period was selected. For example, extending it to two minutes would result in the duty cycles of 0.18%, 0.25% and 0.13% for the A, base and C motes respectively (the communication round duration is kept constant, hence the absolute radio on times do not change). In order to relate these numbers to Dozer, we need to note that our test tree topology was fairly basic. Thus, taking into account that the A mote had only one tree child, its duty cycle of 0.18% corresponds to roughly 0.03% for Dozer in a similar scenario.

The results show that although SOSNA can achieve energy efficient operation, the use of a specialised communication protocol can further improve energy consumption. Nonetheless, the current implementation of the SOSNA's middleware is by no means optimal and further improvements are possible. For example, it would be interesting to employ some of the Dozer's ideas, in particular, scheduling of tree children communications and the communication reliability techniques. However, due to a more general nature of SOSNA's programs, there does not seem to be an obvious way to incorporate the Dozer's approach. In particular, the floating duration of the protocol's communication period make it difficult to bound network communication times.

Tree Aggregation

In the TA scenario, motes aggregate data values by sending them towards the base station along a three-hop static cluster topology (a tree). Multi-hop stream aggregation, as we described it in Section 3.2.2, is performed in n neighbourhood data aggregation rounds and in the case of the TA scenario $n = 3$. In the first round, motes from the bottom level of the tree (third level) send their data items to the nodes from the second level. The second round involves motes from the second level sending their data items to their tree parents at the first level, which send their aggregation result to the base station in the last, third round. In each round, the sending motes switch their radios on only for the time required to send the message while the receiving nodes switch their radios on for the entire round's duration. Although the duration of one application round is known at compile time, sending a message involves a random delay, hence it cannot be predicted. We measured, therefore, the total amount of time mote radios were switched on in each application cycle ($20s$) and Figure 5.13 presents its distribution for motes from all tree levels, for data collected throughout the whole experiment (approximately twelve hours).

The results closely correspond to the expected application behaviour. The base station only receives data, hence it needs to switch its radio on only for the duration of one application round which, in this case, was $500ms$. The graph, however, indicates that the mote's radio was on either for $500ms$ or $1000ms$. This is caused by the operation of the time synchronisation protocol which made each mote switch its radio on for the duration of the service round (also $500ms$), every four cycles. The times for the level-1 and level-2 motes are similar because in both cases the motes aggregate data items and forward the results to their parents. Hence, their radios should be on for exactly $500ms$ plus the random delay of maximum $500ms/3 = 166ms$ and plus the sending time. We can see that in both cases, the maximum radio on time is $700ms$ or $1200ms$, when the time synchronisation is performed. Finally, the times for level-3 motes are the lowest since the motes only send their data items to their tree parents. As expected, the times comprise either the random sending delay or the the sending delay plus $500ms$ for time synchronisation.

Figure 5.13 presents radio operation times for only four nodes out of sixteen. This is because the results are very similar for all motes that are at the same tree level. The figure, therefore, presents the results for arbitrarily chosen motes (one from each tree level) and the values are representative of the whole network.

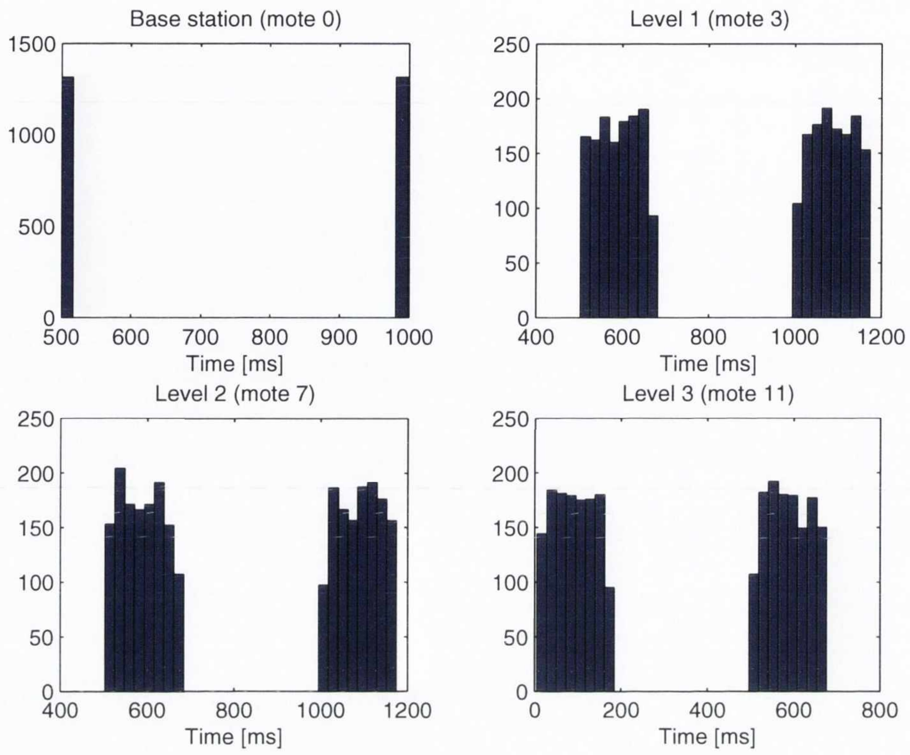


Figure 5.13: Distribution of radio operation times for motes at different tree levels in the TA scenario.

As we can see, the network behaves in a very predictable way, hence knowing the distribution of the message sending times, it is possible to estimate some of its operational characteristics at compilation time. For example, we can use the values presented in Figure 5.13 to approximate the overall node duty cycle. Using the average values of the radio on times and recalling that the application cycle was $20s$ long, we can approximate the duty cycles of the nodes on each of the tree levels to be 0.038, 0.043, 0.043 and 0.018 respectively. This means that all network nodes keep their radios off for over 95% of time and we can relate this value to the battery power consumption rate when we note that the lowest values presented in Figure 5.12 correspond to similar duty cycles. Also, because the network operates in a predictable way, we can see that there is space for further improvement of its energy consumption: if we limited the application round duration to $250ms$ the node duty cycles would have been approximately 0.013, 0.024, 0.024 and 0.011 for each of the tree levels respectively. Naturally, increasing the duration of the sleep round further improves the application's energy efficiency.

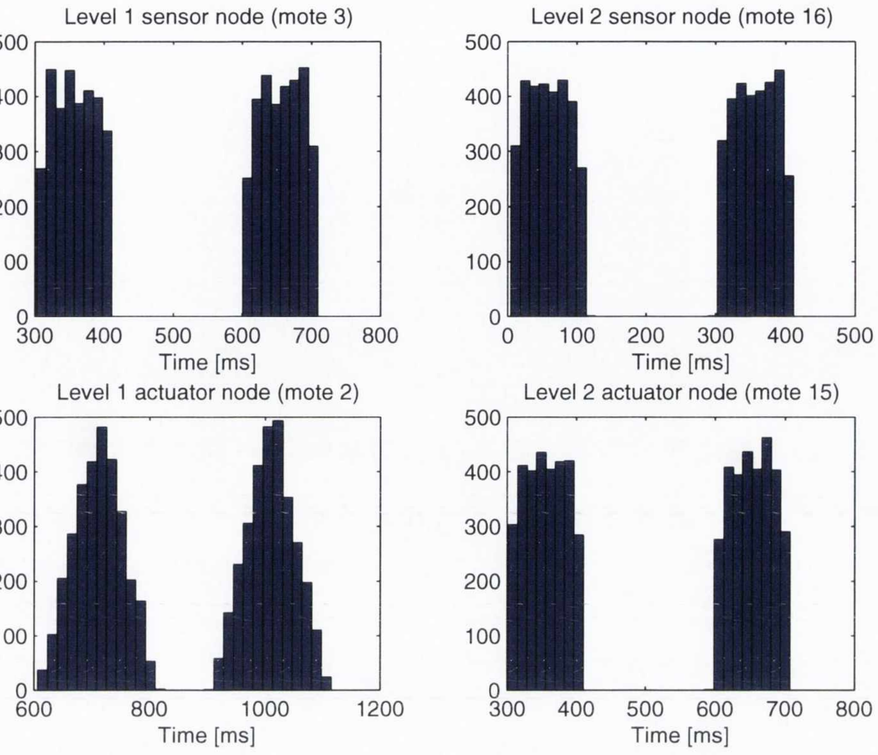


Figure 5.14: Distribution of radio operation times for motes at different tree levels in the DA scenario.

Distributed Actuation

The distributed actuation scenario is similar to the TA scenario in the sense that the motes communicate along the static cluster topology. We use, therefore, the same reasoning to explain the distribution of mote radio utilisation times for different groups of network nodes. The results for the sensor and actuator nodes located at different tree levels are presented in Figure 5.14 while Figure 5.15 presents analogous values for one of the controllers. Similarly, to the TA scenario, the values are concentrated around two time points that represent application cycles with time synchronisation and without it.

The behaviour of the sensor nodes was very similar to the behaviour of the motes in the TA scenario (see Figure 5.13) because in both cases they were simply aggregating values along the static tree topology. The only difference was in the duration of the application round and the in the fact that the TA scenario used three hop-deep cluster topologies. The DA scenario uses the application round duration of $300ms$ hence the sending delay can take maximally $100ms$. As a result, the level-1 sensor nodes switch their radios on, in application cycles without time synchronisation, exactly for the duration of one round (to receive data items from level-2 sensor nodes) increased by the time needed

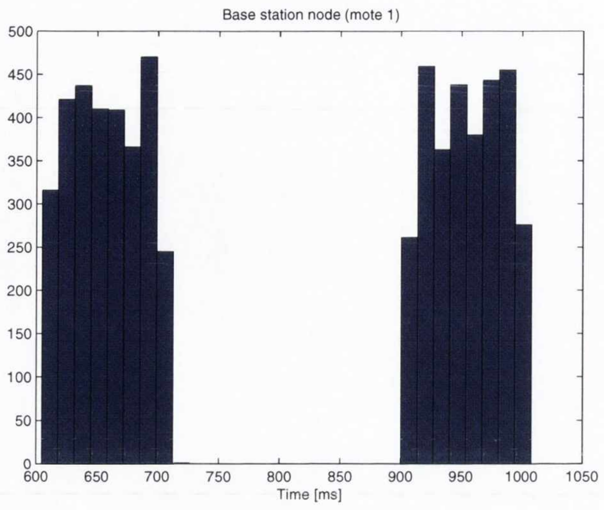


Figure 5.15: Distribution of radio operation times for the base station mote in the DA scenario.

to send a message (little more than 100), while the time synchronisation adds additional $300ms$ of radio utilisation per application cycle. The level-2 sensor nodes, on the other hand, need only to send one message in every application cycle resulting in the radio utilisation times being in the order of either $100ms$ or $400ms$ depending on the cycle.

The operation of actuator nodes is slightly more sophisticated because the nodes engage both in data aggregation and in data propagation tasks. Figure 5.4 shows the communication scheduling for the actuator nodes and we can see that the operations are scheduled for execution one after another. The level-1 actuator nodes, therefore, in every cycle listen for data items from the level-2 actuator nodes, send the aggregation results to the base station, listen for data items from the base station and forward the received values down to level-2 actuator motes. Hence, in cycles without time synchronisation, the level-1 actuator nodes switch their radios on for the duration of two application rounds plus double the time of sending one message, resulting in the radio utilisation times in the range of $600ms$ to $800ms$. The results for level-2 actuator nodes are practically the same as for level-1 sensor nodes because in every cycle both classes of nodes spend one round on listening for data items and send exactly one message.

As regards the controllers, in every cycle they listen for data items from both sensor and actuator nodes (the same round) then, they share the received values among each other (this round includes both sending a message and listening for a message from the other controller) and, finally, they send the control command to the level-1 actuator nodes. Hence, their radios need to be switched on for the duration of two application rounds plus the time to send one message.

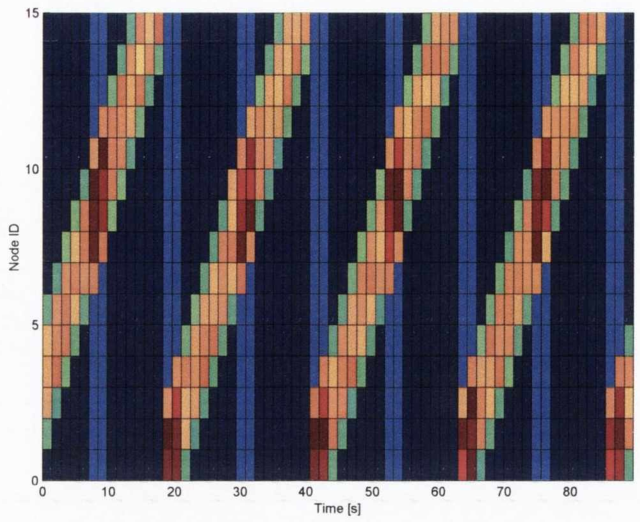


Figure 5.16: Network radio operation times in the TT scenario.

Target Tracking

The target tracking scenario is radically different from the previous two scenarios because it defines a dynamic application whose operation is driven by external events, hence it cannot be predicted. The external events, i.e., the sequence of the physical object's positions drives the network's behaviour by engaging the motes that are located in the object's proximity in the collaborative estimation the object's position. The TT scenario uses region-based stream filtering so that nodes that are not sensing the object can stay dormant, i.e., they can keep their radio transceivers off. As a result, the exact patterns of the motes' radio operation can no longer be predicted at compile time unless we know the exact trajectory of the object. We decided, therefore, to generate the trajectory computationally in a deterministic way so that the experiment can be repeatable and we can easier relate the motes' behaviour to what is expected given the current state of the environment.

We define the object's trajectory in such way that the object periodically visits all of the motes one by one, following their communication topology. The motes compute the current object's position and based on the current distance to the object they either stay dormant or they engage in the leader election process that chooses the mote that is currently the closest to the object, i.e., the one at which the object currently is "residing". Then, the local leader aggregates data from all of its dynamic cluster members and sends the aggregate value to the base station which is one hop away in the communication graph.

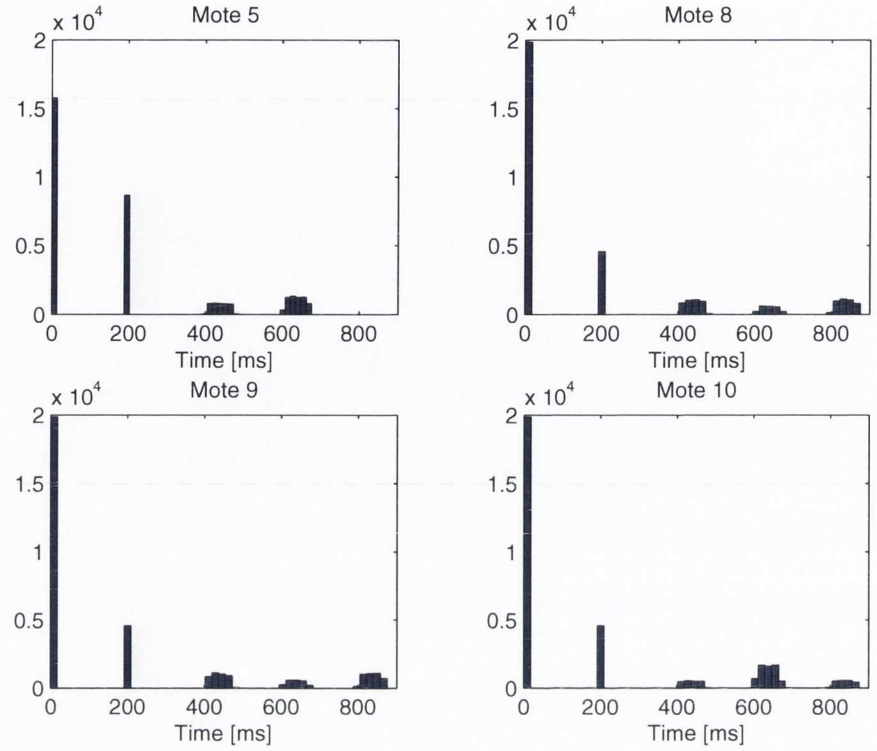


Figure 5.17: Distribution of radio operation times for selected motes in the TT scenario.

Because the network operates synchronously, it is possible to visualise the object's trajectory by plotting the radio operation times of all of the motes at the same time for certain number of application cycles. Therefore, before we analyse the exact radio utilisation times, we discuss the overall network behaviour based on Figure 5.16 that depicts the operation of the whole network for the duration of sixty four cycles. The horizontal axis represents the real time of the experiment while the vertical axis represents the physical locations of the motes (recall from Figure 5.6 that the motes are aligned linearly both in terms of their physical placement, as well as, in terms of their connectivity). The colours represent the amount of time a mote's radio was switched on in the given cycle - the warmer the colour, the longer the radio was being used for. The dark blue colour represents the zero radio use, i.e, motes staying dormant while vertical, light blue lines correspond to the time synchronisation periods. The figure clearly shows the expected network activity pattern as the object repetitively moves from the first mote (mote 0) to the last one.

The plot shows a number of interesting properties. Firstly, we can see that the network is capable of a low-power operation when there is no object to track. Secondly, we can see that the dynamic cluster leaders (nodes in the middle of the detection stripe) incur the highest energy cost because, in addition to participating in the leader election and aggregation processes, they also send the aggregation result to the base station. Finally, the figure shows that uniform, network-wide energy consumption can only be achieved when the object movement patterns uniformly cover the whole network area because otherwise some nodes will inevitably experience higher load. The figure shows, for example, that due to the alignment of the object's movement with the time synchronisation period (eight cycles) the motes number 1,2,9 and 10 happen to have the highest energy consumption.

The exact values of the radio utilisation times are presented in Figure 5.17. We choose four motes for comparison: mote 5, whose activity periods do not overlap with the time synchronisation periods and three other motes whose activity periods do overlap with the time synchronisation periods but in each case in a different way. We do not provide measurements for the base station mote because its operation is deterministic, hence we consider it outside of scope of this scenario. The results closely match the expected values but, in order to explain them we need to discuss radio duty cycling without taking the time synchronisation into consideration.

In each cycle, the behaviour of every mote depends on its distance to the tracked object. Motes remain dormant when they are not detecting the object, i.e., when they are more than two hops away from it, and they do not switch their radios on at all. When motes detect the object, they engage in the leader election process that, in the TT scenario, takes two application rounds to finish because the maximum dynamic cluster depth parameter was set to two hops. After the leader is detected and, hence, the dynamic cluster topology is established, the motes perform stream aggregation along it and the use of the radio transceiver is determined by the mote's position within the cluster. Similarly to aggregation over static cluster topologies, the bottom-level motes send a single message and then switch their radios off, while the level-1 motes spend one round on listening for data items from the bottom-level motes and then send a message to the leader (the cluster head). The leader, on the other hand, listens for data items from the level-1 motes and then sends the result to the base station, hence it has a similar radio utilisation pattern to the level-1 motes. Therefore, given that the application round is $200ms$ long, we can see that motes that are two hops away from the object switch their radios for the duration of two application rounds plus the sending time, resulting in the range of $400-500ms$ of radio utilisation in the whole cycle ($1400ms$). The motes that are one hop away from the object have the same radio utilisation pattern as the detection leader, i.e., three application rounds plus the sending time resulting in the range of $600-700ms$. These values are evident in the radio utilisation results for the mote 5. The additional spike for the value of $200ms$ corresponds to the application cycles in which the mote was participating in time synchronisation. The results for the other three motes follow the same pattern but the values are additionally increased by the coincidence of time synchronisation which adds extra $200ms$.

5.3.2 Synchronisation of Sensing and Actuation

The TA scenario implements an application that computes the maximum spread in sensor sampling times throughout the entire network. The results were collected by the base station node that forwarded them via the serial link to the PC. The application uses the `time` operator to obtain the sensor sampling time and the returned value represents the global time at which the sensor driver was invoked by the application code. The sensor sampling time, therefore, may include, besides the time synchronisation error, additional delays induced by TinyOS. Figure 5.18 presents the maximum spread in sensing times as computed by the aggregation process and reported to the PC by the base station mote. The figure shows that with high probability all motes sample their sensors within nine milliseconds (Figure 5.11 shows that the computed results are representative of the whole network because, in most application cycles, all data values were received).

If we compare the recorded values with the results of the cycle synchronisation error presented in Section 5.2, we can see that the synchronisation errors are likely to dominate the overall imperfections in sensor sampling times because the errors were confined with high probability to the range of $-5ms$ to $5ms$. Similar values were obtained for the maximum spread in actuation times, as measured during the execution of the DA scenario, see Figure 5.19. This result can also be seen as further evidence that the applications conform to the global communication schedule because none of the application rounds introduces additional delays, hence the rounds are likely to be executed within their scheduled time slots.

5.3.3 Memory Consumption

In Chapter 1, we said that WSANs may comprise devices with high resource constraints. The previous section discussed the energy-efficiency of SOSNA applications running on mote networks and it showed that the language can be used to develop applications with scarce energy budgets. Memory is another resource that might be severely limited on some hardware platforms. Table 4.1 provides an overview of hardware capabilities of some of the currently most popular sensor network platforms and it shows, for example, that the TelosB platform has only 48kB of program memory while the MicaZ platform has only 4kB of RAM memory. These values put serious constraints on the set programming tools that can be used for application development because such popular techniques like garbage collection or object orientation cannot be directly used and require optimisation.

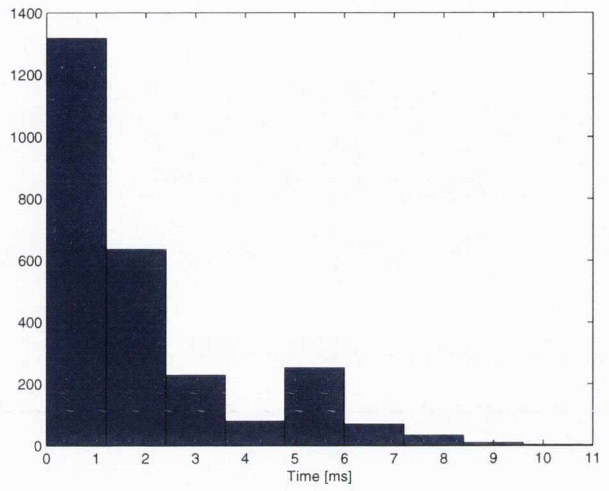


Figure 5.18: Maximum spread in sensing times in the TA scenario.

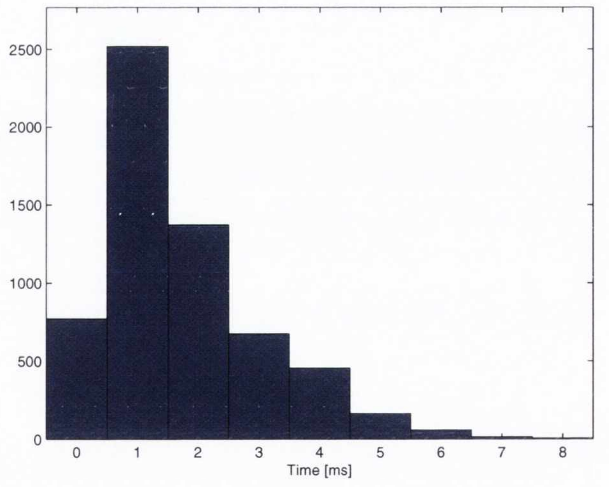


Figure 5.19: Maximum spread in actuation times in the DA scenario.

	Complete		**No Logging**		**Plain**	
Application	RAM	ROM	RAM	ROM	RAM	ROM
TA_{base}	1170	36834	919	26288	657	22792
TA_{motes}	954	38388	780	32364	691	27572
DA_{base}	1238	44708	987	34154	725	30772
DA_{sensor}	952	36246	778	30230	691	25456
$\text{DA}_{actuator}$	982	39556	729	28974	729	28974
TT_{base}	1162	35068	911	24530	629	21344
TA_{motes}	962	40530	788	34522	703	29738

Table 5.1: Memory usage of SOSNA programs compiled for the TelosB target platform.

The SOSNA compiler generates NesC code which then needs to be compiled by the NesC compiler in order to build the target platform executable file. NesC is a programming language that targets the wireless sensor network application domain and whose compiler performs extensive optimisations in order to minimise the size of the executable code. The language offers a static memory management model that requires that all program memory must be statically allocated and no dynamic allocation is possible, unless implemented by the application itself. NesC is the basis of the TinyOS operating system which is a collection of NesC components that are linked against the application code at compilation time in order to build the executable code containing both the application and the minimum of the required TinyOS components.

All three SOSNA programs that implemented the three application scenarios described in Section 5.1.1 were compiled for the TelosB platform, as well as, for the MicaZ mote platform. Table 5.1 and Table 5.2 present the memory usage values that were reported by the NesC compiler after compilation for the TelosB and MicaZ platforms respectively. The values in the RAM column represent the amount of memory allocated for all program data and variables as determined and statically allocated by the compiler while the ROM column represents the compiled program size without taking into consideration program data.

Each of the scenarios defines a heterogeneous application that comprises either two or three types of devices. The SOSNA compiler generates a separate program for each of them and these programs need to be separately compiled into different executables. We refer to the device-specific parts of the global SOSNA macro-program by using scenario names subscripted with the corresponding device class name, as defined in program listings in Section 5.1.1. In case of the TelosB platform, the programs were compiled in three different settings. The "Complete" column represents memory usage values that include all software components used in the experiments, i.e., the Flash logging component, as well as, the sensor and actuator drivers. Because these components are based largely on TinyOS

	No Logging		**Plain**	
Application	RAM	ROM	RAM	ROM
TA_{base}	1115	29580	875	26230
TA_{motes}	-	-	903	32894
DA_{base}	1178	40360	938	37130
DA_{sensor}	-	-	895	30230
$\text{DA}_{actuator}$	933	34386	933	34386
TT_{base}	1096	27372	856	24322
TA_{motes}	-	-	908	35852

Table 5.2: Memory usage of SOSNA programs compiled for the MicaZ target platform.

libraries the resulting program sizes do not give the true picture of the overhead the SOSNA-generated code. Therefore, all programs were additionally compiled with the logging switched off (the second column), as well as, substituting a simple sensor and actuator stubs in place of the `TelosLight` sensor driver and the `Report` actuator driver (the last column). The similar approach was taken in the case of the MicaZ compilation target but since the logging component, as of yet, does not support this platform, Table 5.2 gives only memory usage values for the "No logging" and "Plain" settings. Also, because the `TelosLight` sensor driver is platform specific, the table provides only the "Plain" setting values for those programs that use this component (TA_{motes}, DA_{sensor} and TT_{motes}).

The results show that SOSNA programs can be compiled to small executables and that they make efficient use of RAM memory. The results also show that support libraries take up non-negligible amount of the program memory putting additional constraints on the size of the application code. The tables show a significant difference in the memory usage for the two platforms. We believe that this is due to the fact that NesC internally uses a different C compiler to generate the binary code for each of the platforms and that the compilers may differ in the range of code optimisations they perform, as well as, that the target hardware architecture and the microcontroller's instruction set may have different effect on the binary code sizes.

The last column of each of the tables is the most representative in terms of the memory overhead induced by the SOSNA application code. The results follow a clear pattern that relates the resulting program size to the number of data aggregation tasks the program implements. We can ignore the effect of other factors such as the scalar functions or the stream expressions evaluated by each of the programs because the three scenarios used minimal data processing, hence the communication expressions dominate the program sizes. Because all SOSNA's communication expressions are decomposed into sequences of neighbourhood data aggregation operations, the size of the device-specific programs is dominated by the number of neighbourhood data aggregation operations they implement. Hence,

Application	TelosB		MicaZ	
	RAM	ROM	RAM	ROM
NesC RadioCntToLeds	320	11526	300	11398
SOSNA Minimal	601	19700	823	21920
SOSNA 1-hop Aggr.	607	20152	834	22512
SOSNA 2-hop Aggr.	655	21764	858	24786
SOSNA 3-hop Aggr.	637	22668	850	25858
SOSNA 4-hop Aggr.	637	23744	850	27184
SOSNA 5-hop Aggr.	637	24830	850	28540

Table 5.3: Comparison of memory usage for a basic NesC application, a basic SOSNA application and SOSNA applications implementing stream aggregation along static cluster topologies of different depths.

the base station-specific programs are generally the smallest when compiled in the "Plain" setting.

This view is further supported by the results presented in Table 5.3 that compares the memory usage values of a simple SOSNA program implementing multi-hop stream aggregation over the static cluster topology. As we said in Chapter 3, the maximum depth of static cluster topologies is a compilation parameter that naturally influences the size of the application code because deeper topologies result in more neighbourhood aggregation rounds needed to compute the final result. The table shows that extending the topology by one level results in over 1kB of memory overhead. This value, however, represents the memory cost of a single neighbourhood aggregation round, i.e., it is related to the aggregation of a single stream expression. Therefore, two independent aggregation expressions, even when scheduled for execution in parallel, will result in double of the cost. This is a deficiency of our prototype compiler that generates an independent handler for each *data item* a program receives. Different data items directly represent the values of independently aggregated streams. Therefore, even if several data items are packed into a single message, each of them is independent and it receives a dedicated handler. The prototype compiler generates a very similar code of each handler resulting in repetitions that lead to the increased memory cost. One of the extensions we consider for the future work, is to reduce this cost by making message handling more dynamic by using a generic handler definition that is parametrised to handle data items of different types.

The Table also shows memory usage values for two additional applications. The first one, RadioCntToLeds, is a simple NesC application that is distributed with TinyOS and that periodically sends and receives radio packets containing increasing values of an integer counter. The application does not implement any data processing, hence its memory usage roughly represents the memory cost of the TinyOS's radio stack, the LEDs driver and of a single millisecond timer. Because all SOSNA applications include these components, the values can be used to approximate the total memory cost of the SOSNA middleware together with the application code. The second application, SOSNA Minimal, is a primitive program that outputs a constant integer stream to a simple actuator. The program's memory usage corresponds roughly to the memory cost of the basic TinyOS components and of the SOSNA middleware (hence also the time synchronisation protocol). The values can be used, therefore, as the basis for the estimation of the real performance of the SOSNA compiler in terms of the memory requirements of its programs. For example, the real memory cost of the target tracking application TT_{motes} is in the order of 100 bytes of RAM memory 10kB of program memory, when compiled for the TelosB platform while, for the largest of all presented programs - DA_{base}, the real memory cost of the application code is approximately 130 bytes of RAM memory and 21kB of program memory.

5.4 Discussion

This section presented the evaluation of operational characteristics of SOSNA programs running in mote networks. Based on the generic application scenarios that implemented three typical WSN/WSAN applications, we showed that SOSNA applications operate in a timely manner and that their execution conforms to the abstract synchronous macro-programming model. We showed that the global communication schedule imposed by the model allows to implement duty cycling policies that result in significant savings in terms of battery power consumption, as well as, they allow to develop WSAN applications that require synchronisation of sensing and actuation among network nodes. Finally, we analysed the memory cost of SOSNA applications and showed that the prototype compiler can generate efficient application code that can be run on resource constrained hardware platforms while leaving memory space for additional software components such as sensor or actuator drivers.

All experiments were performed on a network of seventeen TelosB and MicaZ motes deployed in an office environment. Although such a configuration demonstrates many important characteristics of a real wireless sensor network, it certainly does not validate the model in all conceivable deployment scenarios. In fact, the scale factor often manifests itself when larger networks are considered, increasing hardware and communication failure rates, node reboots, loss of time synchronization, as well as the wireless channel load. The prototype SOSNA middleware can handle transient node, communication and time synchronization failures, and Section 4.2.3 discussed the limitations, as well as prospects for improvement of the employed techniques. Nevertheless, it would be desirable to evaluate SOSNA applications in more general and realistic scenarios, i.e., running on larger networks and over longer periods of time. A possible step in this direction would be to use one of the existing WSN testbeds, e.g., the Harvard University's Motelab (http://motelab.eecs.harvard.edu/). We put this, however, among the plans for future work along with the evaluation of alternative SOSNA middleware designs that would incorporate reliable communications and improved duty cycling strategies.

Chapter 6

Conclusions and Future Work

This thesis presented the synchronous macro-programming model that merges the synchronous programming paradigm with the macro-programming paradigm for the development of energy-efficient and timely WSAN applications. In order to validate the model, we designed and implemented the synchronous macro-programming language SOSNA which is a stream-based functional formalism targeting heterogeneous WSAN applications and whose compiler is capable of generating application code that automatically duty cycles operation of the radio transceiver. We evaluated the timeliness and energy-efficiency properties of SOSNA applications by means of running a range of generic applications on wireless networks of resource-constrained devices, commonly referred to as motes. We showed that the applications operated in a timely and predictable fashion, as well as, that the radio duty cycling policies, implemented by the prototype compiler and enabled by the synchronous macro-programming model, lead to significant decrease in battery power consumption. Also, we showed that the compiler was capable of generating memory-efficient application code that fits on resource-constrained hardware platforms. In this section we overview the contributions of this work and discuss prospects for future research.

6.1 Contribution

The core difficulty in programming wireless sensor-actuator networks is related to the need to orchestrate sensor data processing and actuation in a distributed and timely manner while taking into consideration the possibly resource-constrained nature of the underlying hardware. The overall contribution of our work is to show that the synchronous programming paradigm can be blended with the macro-programming paradigm for the development of energy-efficient and timely WSAN applications.

Macro-programming is a programming paradigm originally proposed for WSN/WSAN applications in which the network behaviour is specified using constructs whose semantics spans potentially over large collections of network nodes. Macro-programming languages allow to concisely specify complex interactions among network nodes thus facilitating the programming task. We show that by giving synchronous semantics to a macro-programming language the compiler is able to schedule all network communications and to generate application code capable of duty cycling radio operation.

The central assumption of the model is such that the network must be time synchronised in order for the global communication schedule to be implemented, as well as, that the extent of all network communications must be bounded so that the predictability of system response times can be ensured. In Chapter 1, we discuss the core characteristics of WSANs and motivate our design choices by arguing that duty cycling the radio transceiver is crucial to ensuring energy-efficient operation of networks comprising resource-constrained devices while the reliability of long-range multi-hop radio communication was shown to be below acceptable levels in large networks. We also show that the time synchronisation requirement is realistic given the current state of the art in the area of time synchronisation protocols for wireless sensor networks.

Chapter 2 reviewed the existing programming languages for wireless sensor and wireless sensor-actuator networks and we concluded that none of the languages that targeted general-purpose WSAN applications, i.e., which allowed to express sensor data transformations and actuation, provided support for both timely execution and radio duty cycling.

In Chapter 3 we formalised the synchronous macro-programming model in terms of neighbourhood data aggregation rounds which are the basic building blocks of high-level communication primitives that may be defined by languages implementing the model. Then, we presented the design of the SOSNA programming language and we showed that such high-level macro-programming constructs as multi-hop stream aggregation and propagation, leader-based tree topology construction or region-based computations can be defined in terms of the synchronous macro-programming model, i.e., as sequences of neighbourhood data aggregation rounds, thus giving evidence to the model's applicability to WSAN application development. The chapter also discusses how network heterogeneity, which is an inherent characteristic of many WSANs, can be handled in the synchronous macro-programming setting. We showed that type inference and minimal program annotations can be used to automatically discover the partitioning of the global heterogeneous program into a set of device specific programs. Finally, the chapter discusses crucial language design choices such as disallowance of stream function recursion to ensure the compliance with the synchronous macro-programming model.

The implementation of a prototype SOSNA compiler is discussed in Chapter 4. The main purpose of the chapter is to demonstrate a set of techniques that can be used for efficient implementation of a synchronous macro-programming language. We described, therefore, the most important program transformation stages and showed how the abstract language semantics are translated into concrete scheduling and radio duty cycling policies. The second part of the chapter describes how SOSNA applications are executed in mote networks. We showed how the application schedules can be implemented in NesC, the core language of the popular TinyOS operating system, as well as, how network time synchronisation is maintained within the frame of the global communication schedule.

We concluded the thesis with the evaluation of timeliness and energy efficiency of SOSNA applications running on wireless mote networks. We built the evaluation around three generic application scenarios that served both as proof of concept implementations and as a basis for measuring of a number of application execution parameters. We ran the applications on networks of TelosB and MicaZ motes showed that our approach results in timely and predictable application behaviour and that it leads to measurable benefits in terms of battery power consumption. Finally, we analysed the memory overhead of the applications compiled for these hardware platforms and we showed that the prototype SOSNA compiler is capable of generating memory-efficient application code that fits on resource-constrained hardware.

6.2 Future Work

There are many possible directions for future work. The SOSNA programming language described in this thesis is intended a demonstration of the applicability of the synchronous macro-programming model, hence its design is open. The model, on the other hand, is general and can accommodate many extensions. We list therefore a number of interesting research avenues stemming from our work.

Reliable communications are a core requirement of real system applications. Therefore, it is necessary to experiment with alternative SOSNA middleware designs that would provide better reliability guarantees. A number of possible directions are possible following, for example, the ideas used in the design of the Wireless HART protocol, i.e., fine grained communication scheduling, retransmissions, synchronous acknowledgements, channel hopping and channel black-listing. Also, maintenance of stable network topologies is an important aspect of any reliability infrastructure.

Real-time operation is a natural extension of the work. The prototype SOSNA compiler generates NesC code for execution in TinyOS which, unfortunately, is not a real-time operating system. It would be interesting, therefore, to experiment with compilation targets that offer some sort of timeliness guarantees. Also, the use of a different time synchronisation protocol, alternative scheduling strategies or more advanced program optimisation techniques are problems well worth investigating.

Heterogeneity support is another interesting research direction. So far, we considered only heterogeneity in processing hardware while assuming that all network nodes are able to communicate with each other. It is often the case, however, that different communication media are used to exchange information between devices of different classes. For example, in multi-tiered networks sensor nodes use low-power radio communications while the base stations might use more powerful means, including wire-line communications. How can we accommodate heterogeneity in communication media in the language without cluttering its syntax with unnecessary annotations?

Alternative language designs are also worth considering. Is it possible to reconcile multi-rate streams with the synchronous macro-programming model? Clearly, some applications might require high-rate stream processing and the current language definition pushes it down to the implementation of the sensor drivers. Possible directions might include extensions to the language's type system and a form of clock calculus to guarantee the safety of stream composition (Pouzet 2006) or the approach taken by Newton et al. (2008) who propose to use stream segmentation as a way of tackling the communication and the computational cost of high-rate stream processing.

Finally, it might be useful to consider a more general language formulation that allows the programmer to consisely define all SOSNA's high-level communication primitives rather than having them hard coded in the language definition. Such design could perhaps extend the language's applicability or expressiveness.

Appendix A

Sosna Machine Instruction Set

`alloc` *type*

Allocates memory on the heap, places a pointer to it on the stack.

`free` *type*

Deallocate the memory from the heap top, remove the top stack pointer.

`pushNull`

Put a null pointer on the stack.

`pushVal` *v type*

Allocate memory for the data value, leave pointer to it on the stack.

`pushRef` *i*

Push copy the pointer located at the i-th position on the stack (from the bottom).

`pushPreg`

Push the current value of the pointer register PREG.

`pop` *n*

Remove the top-level stack pointer.

`loadPreg`

Set the value of the PREG pointer register to the value of the top stack pointer.

`clearPreg`

Set the PREG register to null.

`saveHeap`

Push the pointer to the heap top to the stack.

`restoreHeap`

Set the heap top pointer to the value of the top stack pointer. Pop it from the stack.

`store` d_i p_i *type*

Copy the data value pointed by the top stack pointer to the i-th global heap location.

`present?` n

Allocate a Boolean value on the heap, push the pointer to it to the stack and set it to true if the top-n pointers are not null.

`call` f

Invoke the function f with the pointers to its arguments at the top of the stack.

`outp` *id*

Invoke the actuator id passing it the value pointed by the top stack pointer.

`halt`

Finish the round (implemented as a radio ON/OFF command)

`nop`

No operation.

`osCall` *proc* $arg_1 \ldots arg_n$ *handler*

Call an operating system routine passing a result handler.

`initMsg` *id*

Clear the message buffer's data item presence bit field.

`txrx?`

Check whether the current data item can be sent.

`tt − level?`

Check whether the topology descriptor pointed by the top stack pointer matches the required topology level.

`loadMsgItem` *id n*

Store a value as one of teh message buffer's data items.

`pushMsgItem` *id n*

Push the message buffer's data item to the stack and heap.

Bibliography

Abelson, H. & Sussman, G. J. (1996), *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, USA.

Akyildiz, I. F. & Kasimoglu, I. H. (2004), 'Wireless sensor and actor networks: research challenges', *Ad Hoc Networks* **2**(4), 351–367.

Arora, A., Ramnath, R., Ertin, E., Sinha, P., Bapat, S., Naik, V., Kulathumani, V., Zhang, H., Cao, H., Sridharan, M., Kumar, S., Seddon, N., Anderson, C., Herman, T., Trivedi, N., Zhang, C., Nesterenko, M., Shah, R., Kulkarni, S. S., Aramugam, M., Wang, L., Gouda, M. G., ri Choi, Y., Culler, D. E., Dutta, P., Sharp, C., Tolle, G., Grimmer, M., Ferriera, B. & Parker, K. (2005), Exscal: Elements of an extreme scale wireless sensor network., *in* 'RTCSA', IEEE Computer Society, pp. 102–108.

Athans, M. & Falb, P. (2006), *Optimal Control: An Introduction to the Theory and Its Applications*, Dover Publications.

Awan, A., Jagannathan, S. & Grama, A. (2007), 'Macroprogramming heterogeneous sensor networks using cosmos', *SIGOPS Oper. Syst. Rev.* **41**(3), 159–172.

Bakshi, A., Prasanna, V. K., Reich, J. & Larner, D. (2005), The abstract task graph: a methodology for architecture-independent programming of networked sensor systems, *in* 'EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services', USENIX Association, Berkeley, CA, USA, pp. 19–24.

Beal, J. & Bachrach, J. (2006), 'Infrastructure for engineered emergence on sensor/actuator networks', *Intelligent Systems, IEEE* **21**(2), 10–19.

Beal, J. & Sussman, G. (2005), Biologically-inspired robust spatial programming, AI Memo MIT-CSAIL-TR-2005-003, Massachusetts Institute of Technology.

Benveniste, A., Caillaud, B. & Guernic, P. L. (1999), From synchrony to asynchrony, *in* 'CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory', Springer-Verlag, London, UK, pp. 162–177.

Benveniste, A., Caspi, P., di Natale, M., Pinello, C., Sangiovanni-Vincentelli, A. & Tripakis, S. (2007), Loosely time-triggered architectures based on communication-by-sampling, *in* 'EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software', ACM, New York, NY, USA, pp. 231–239.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P. & de Simone, R. (2003), 'The synchronous languages 12 years later', *Proceedings of the IEEE* **91**(1), 64–83.

Berry, G. & Cosserat, L. (1985), The esterel synchronous programming language and its mathematical semantics, *in* 'Seminar on Concurrency, Carnegie-Mellon University', Springer-Verlag, London, UK, pp. 389–448.

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A. & Han, R. (2005), 'Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms', *Mob. Netw. Appl.* **10**(4), 563–579.

Bischoff, U. & Kortuem, G. (2007), A compiler for the smart space, *in* 'Ambient Intelligence, European Conference, AmI 2007, Darmstadt, Germany', pp. 230–247.

Bonenfant, A., Ferdinand, C., Hammond, K. & Heckmann, R. (2007), Worst-case execution times for a purely functional language, *in* 'Proc. Implementation of Functional Languages (IFL 2006)', Vol. 4449 of *Lecture Notes in Computer Science*, Springer.

Boussinot, F. & Susini, J.-F. (1998), 'The sugarcubes tool box: a reactive java framework', *Softw. Pract. Exper.* **28**(14), 1531–1550.

Brown, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T. & Spindel, M. (2007), 'The virtual node layer: A programming abstraction for wireless sensor networks', *SIGBED Rev.* **4**(3), 7–12.

Burri, N., von Rickenbach, P. & Wattenhofer, R. (2007), Dozer: ultra-low power data gathering in sensor networks, *in* 'IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks', ACM, New York, NY, USA, pp. 450–459.

Butera, W. J. (2002), Programming a paintable computer, PhD thesis. Supervisor-Bove,Jr., V. Michael.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S. & Niebert, P. (2003), From simulink to scade/lustre to tta: a layered approach for distributed embedded applications, *in* 'LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems', ACM, New York, NY, USA, pp. 153–162.

Caspi, P., Girault, A. & Pilaud, D. (1999), 'Automatic distribution of reactive systems for asynchronous networks of processors', *IEEE Trans. Softw. Eng.* **25**(3), 416–427.

Caspi, P., Pilaud, D., Halbwachs, N. & Plaice, J. A. (1987), Lustre: a declarative language for real-time programming, *in* 'POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM, New York, NY, USA, pp. 178–188.

Caspi, P. & Pouzet, M. (1996), Synchronous kahn networks, *in* 'ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming', ACM, New York, NY, USA, pp. 226–238.

Cheong, E. (2007), Actor-Oriented Programming for Wireless Sensor Networks, PhD thesis, EECS Department, University of California, Berkeley.

Chu, D., Popa, L., Tavakoli, A., Hellerstein, J. M., Levis, P., Shenker, S. & Stoica, I. (2007), The design and implementation of a declarative sensor network system, *in* 'SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 175–188.

Crossbow Technology (2009).
URL: *http://www.xbow.com/*

Dalton, A., McCartney, W., GhoshDastidar, K., Hallstrom, J., Sridhar, N., Herman, T., Leal, W., Arora, A. & Gouda, M. (2008), Desal alpha: An implementation of the dynamic embedded sensor-actuator language, *in* 'Computer Communications and Networks, 2008. ICCCN '08. Proceedings of 17th International Conference on', pp. 1–7.

D'Andrea, R. & Dullerud, G. (2003), 'Distributed control design for spatially interconnected systems', *Automatic Control, IEEE Transactions on* **48**(9), 1478–1495.

Delaval, G., Girault, A. & Pouzet, M. (2008), A type system for the automatic distribution of higher-order synchronous dataflow programs, *in* 'LCTES', pp. 101–110.

Dunkels, A., Grönvall, B. & Voigt, T. (2004), Contiki - a lightweight and flexible operating system for tiny networked sensors, *in* 'Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)', Tampa, Florida, USA.

Dunkels, A., Schmidt, O., Voigt, T. & Ali, M. (2006), Protothreads: Simplifying event-driven programming of memory-constrained embedded systems, *in* 'Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)', Boulder, Colorado, USA.

Elliott, C. & Hudak, P. (1997), Functional reactive animation, *in* 'ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming', ACM, New York, NY, USA, pp. 263–273.

Esterel Technologies (2009), 'Scade suite', http://www.esterel-technologies.com/.

Eswaran, A., Rowe, A. & Rajkumar, R. (2005), Nano-rk: an energy-aware resource-centric rtos for sensor networks, *in* 'Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International', pp. 10 pp.–265.

Flatt, M. & PLT (2009), Reference: PLT scheme, Reference Manual PLT-TR2009-reference-v4.2.2, PLT Scheme Inc.

Fok, C.-L., Roman, G.-C. & Lu, C. (2009), 'Agilla: A mobile agent middleware for self-adaptive wireless sensor networks', *ACM Trans. Auton. Adapt. Syst.* **4**(3), 1–26.

Frank, C. & Römer, K. (2005), Algorithms for generic role assignment in wireless sensor networks, *in* 'SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 230–242.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. & Culler, D. (2003), The nesc language: A holistic approach to networked embedded systems, *in* 'PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation', ACM, New York, NY, USA, pp. 1–11.

Gelernter, D. (1985), 'Generative communication in linda', *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112.

Gnawali, O., Jang, K.-Y., Paek, J., Vieira, M., Govindan, R., Greenstein, B., Joki, A., Estrin, D. & Kohler, E. (2006), The tenet architecture for tiered sensor networks, *in* 'SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 153–166.

Greenstein, B., Kohler, E. & Estrin, D. (2004), A sensor network application construction kit (snack), *in* 'SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 69–80.

Gummadi, R., Gnawali, O. & Govindan, R. (2005), Macro-programming wireless sensor networks using kairos, pp. 126–140.

Gummadi, R., Kothari, N., Millstein, T. & Govindan, R. (2007), Declarative failure recovery for sensor networks, *in* 'AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development', ACM, New York, NY, USA, pp. 173–184.

Han, C.-C., Kumar, R., Shea, R., Kohler, E. & Srivastava, M. (2005), A dynamic operating system for sensor nodes, *in* 'MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services', ACM Press, New York, NY, USA, pp. 163–176.

HART Communication Foundation (2010).
URL: *http://www.hartcomm.org*

He, T., Krishnamurthy, S., Luo, L., Yan, T., Gu, L., Stoleru, R., Zhou, G., Cao, Q., Vicaire, P., Stankovic, J. A., Abdelzaher, T. F., Hui, J. & Krogh, B. (2006), 'Vigilnet: An integrated sensor network system for energy-efficient surveillance', *ACM Trans. Sen. Netw.* **2**(1), 1–38.

He, T., Stankovic, J. A., Lu, C. & Abdelzaher, T. (2003), Speed: a stateless protocol for real-time communication in sensor networks, *in* 'Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on', pp. 46–55.

Heinzelman, W. B., Murphy, A. L., Carvalho, H. S. & Perillo, M. A. (2004), 'Middleware to support sensor network applications', *Network, IEEE* **18**(1), 6–14.

Hnat, T. W., Sookoor, T. I., Hooimeijer, P., Weimer, W. & Whitehouse, K. (2008), Macrolab: a vector-based macroprogramming framework for cyber-physical systems, *in* 'SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems', ACM, New York, NY, USA, pp. 225–238.

Hofmann, M. & Jost, S. (2003), 'Static prediction of heap space usage for first-order functional programs', *SIGPLAN Not.* **38**(1), 185–197.

Hohlt, B., Doherty, L. & Brewer, E. (2004), Flexible power scheduling for sensor networks, *in* 'IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks', ACM, New York, NY, USA, pp. 205–214.

Hudak, P. (1998), Modular domain specific languages and tools, *in* 'in Proceedings of Fifth International Conference on Software Reuse', IEEE Computer Society Press, pp. 134–142.

Karpiński, M. & Cahill, V. (2007), High-level application development is realistic for wireless sensor networks, *in* 'Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on', pp. 610–619.

Kasten, O. & Römer, K. (2005), Beyond event handlers: programming wireless sensors with attributed state machines, *in* 'IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks', IEEE Press, Piscataway, NJ, USA, p. 7.

Kerboeuf, M., Nowak, D. & Talpin, J.-P. (2000), Specification and verification of a steam-boiler with signal-coq, *in* 'Theorem Proving in Higher Order Logics, 13th International Conference', Vol. 1869 of *Lecture Notes in Computer Science*, Springer.

Klavins, E. & Murray, R. (2004), 'Distributed algorithms for cooperative control', *Pervasive Computing, IEEE* **3**(1), 56–65.

Kopetz, H. (1997), *Real-Time Systems : Design Principles for Distributed Embedded Applications (The International Series in Engineering and Computer Science)*, Springer.

Koshy, J. & Pandey, R. (2005), Vmstar: synthesizing scalable runtime environments for sensor networks, *in* 'SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 243–254.

Kothari, N., Gummadi, R., Millstein, T. & Govindan, R. (2007), 'Reliable and efficient programming abstractions for wireless sensor networks', *SIGPLAN Not.* **42**(6), 200–210.

Kwon, Y., Sundresh, S., Mechitov, K. & Agha, G. (2006), Actornet: an actor platform for wireless sensor networks, *in* 'AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems', ACM, New York, NY, USA, pp. 1297–1300.

LeGuernic, P., Gautier, T., Le Borgne, M. & Le Maire, C. (1991), 'Programming real-time applications with signal', *Proceedings of the IEEE* **79**(9), 1321–1336.

Levis, P. (2005), Application Specific Virtual Machines: Operating System Support for User-Level Sensornet Programming, PhD thesis, University of California, Berkeley.

Levis, P., Gay, D. & Culler, D. (2005), Active sensor networks, *in* 'NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation', USENIX Association, Berkeley, CA, USA, pp. 343–356.

Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D. & Wolisz, A. (2005), T2: A second generation os for embedded sensor networks, Technical report, Telecommunication Networks Group, Technische Universität Berlin.

Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E. & Culler, D. (2004), The emergence of networking abstractions and techniques in tinyos, *in* 'NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation', USENIX Association, Berkeley, CA, USA, pp. 1–1.

Li, S.-F. (2006), Wireless sensor actuator network for light monitoring and control application, *in* 'Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE', Vol. 2, pp. 974–978.

Li, S., Son, S. & Stankovic, J. (2004), 'Event detection services using data service middleware in distributed sensor networks', *Telecommunication Systems* **26**, 351–368.

Lin, C., Federspiel, C. C. & Auslander, D. M. (2002), Multi-sensor single-actuator control of hvac systems, *in* 'Proc. of The International Conference for Enhanced Building Operations'.

Liu, H., Roeder, T., Walsh, K., Barr, R. & Sirer, E. G. (2005), Design and implementation of a single system image operating system for ad hoc networks, *in* 'MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services', ACM, New York, NY, USA, pp. 149–162.

Liu, J., Chu, M., Liu, J., Reich, J. & Zhao, F. (2003), 'State-centric programming for sensor-actuator network systems', *IEEE Pervasive Computing* **2**(4), 50–62.

Liu, J., Liu, J., Reich, J., Cheung, P. & Zhao, F. (2003), Distributed group management for track initiation and maintenance in target localization applications, *in* '2nd Workshop on Information Processing in Sensor Networks (IPSN)', pp. 113–128.

Liu, T. & Martonosi, M. (2003), Impala: a middleware system for managing autonomic, parallel sensor systems, *in* 'PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming', ACM, New York, NY, USA, pp. 107–118.

Luo, L., Abdelzaher, T. F., He, T. & Stankovic, J. A. (2006), 'Envirosuite: An environmentally immersive programming framework for sensor networks', *ACM Trans. Embed. Comput. Syst.* **5**(3), 543–576.

Lynch, J. P., Wang, Y., Swartz, R. A., Lu, K. C. & Loh, C. H. (2007), 'Implementation of a closed-loop structural control system using wireless sensor networks', *Structural Control and Health Monitoring* **15**(4), 518–539.

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

Mainland, G., Kang, L., Lahaie, S., Parkes, D. C. & Welsh, M. (2004), Using virtual markets to program global behavior in sensor networks, *in* 'EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop', ACM, New York, NY, USA, p. 1.

Mainland, G., Morrisett, G. & Welsh, M. (2008), Flask: staged functional programming for sensor networks, *in* 'ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming', ACM, New York, NY, USA, pp. 335–346.

Mamei, M. & Nagpal, R. (2007), 'Macro programming through bayesian networks: Distributed inference and anomaly detection', *Pervasive Computing and Communications, IEEE International Conference on* **0**, 87–96.

Mandel, L. & Pouzet, M. (2005), Reactiveml: a reactive extension to ml, *in* 'PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming', ACM, New York, NY, USA, pp. 82–93.

Maraninchi, F., Samper, L., Baradon, K. & Vasseur, A. (2008), 'Lustre as a system modeling language: Lussensor, a case-study with sensor networks', *Electron. Notes Theor. Comput. Sci.* **203**(4), 95–110.

Maróti, M., Kusy, B., Simon, G. & Lédeczi, A. (2004), The flooding time synchronization protocol, *in* 'SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 39–49.

Marrón, P. J., Lachenmann, A., Minder, D., Hähner, J., Sauter, R. & Rothermel, K. (2005), Tinycubus: A flexible and adaptive framework for sensor networks, *in* 'In EWSN'05', pp. 278–289.

Melodia, T., Pompili, D., Gungor, V. C. & Akyildiz, I. F. (2005), A distributed coordination framework for wireless sensor and actor networks, *in* 'MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing', ACM, New York, NY, USA, pp. 99–110.

Montestruque, L. & Lemmon, M. D. (2008), Csonet: a metropolitan scale wireless sensor-actuator network, *in* 'International Workshop on Mobile Device and Urban Sensing (MODUS)'.

Moss, D. & Levis, P. (2008), Box-macs: Exploiting physical and link layer boundaries in low-power networking., Technical Report SING-08-00, Stanford University.

Mottola, L. & Picco, G. P. (2006), Logical neighborhoods: A programming abstraction for . . ., *in* 'In Proceedings of the 2nd International Conference on Distributed Computing on Sensor Systems (DCOSS)'.

Munir, S., Lin, S., Hoque, E., Nirjon, S. M. S., Stankovic, J. A. & Whitehouse, K. (2010), Addressing burstiness for reliable communication and latency bound generation in wireless sensor networks, *in* 'IPSN 2010: Proceedings of the 9th International Conference on Information Processing in Sensor Networks'.

Newton, R., Arvind & Welsh, M. (2005), Building up to macroprogramming: an intermediate language for sensor networks, *in* 'IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks', IEEE Press, Piscataway, NJ, USA, p. 6.

Newton, R., Morrisett, G. & Welsh, M. (2007), The regiment macroprogramming system, *in* 'Proc. 6th International Symposium on Information Processing in Sensor Networks IPSN 2007', pp. 489–498.

Newton, R. R., Girod, L. D., Craig, M. B., Madden, S. R. & Morrisett, J. G. (2008), Design and evaluation of a compiler for embedded stream programs, *in* 'LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems', ACM, New York, NY, USA, pp. 131–140.

Newton, R., Toledo, S., Girod, L., Balakrishnan, H. & Madden, S. (2009), Wishbone: profile-based partitioning for sensornet applications, *in* 'NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation', USENIX Association, Berkeley, CA, USA, pp. 395–408.

Newton, R. & Welsh, M. (2004), Region streams: functional macroprogramming for sensor networks, *in* 'DMSN', pp. 78–87.

Ni, Y., Kremer, U. & Iftode, L. (2004), A programming language for ad-hoc networks of mobile devices, *in* 'LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems', ACM, New York, NY, USA, pp. 1–12.

Nowak, D., Beauvais, J.-R. & Talpin, J.-P. (1998), Co-inductive axiomatization of a synchronous language, *in* 'Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics', Springer-Verlag, London, UK, pp. 387–399.

Oung, R., Bourgault, F., Donovan, M. & D'Andrea, R. (2010), The distributed flight array, *in* 'IEEE Conference on Robotics and Automation (ICRA), 2010, Anchorage,'.

Pierce, B. C. (2004), *Advanced Topics in Types and Programming Languages*, The MIT Press.

Polastre, J., Hill, J. & Culler, D. (2004), Versatile low power media access for wireless sensor networks, *in* 'SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 95–107.

Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S. & Stoica, I. (2005), A unifying link abstraction for wireless sensor networks, *in* 'SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 76–89.

Potop-Butucaru, D., Simone, R. D. & pierre Talpin, J. (2005), 'The synchronous hypothesis and synchronous languages', Embedded Systems Handbook.

Pottie, G. J. & Kaiser, W. J. (2000), 'Wireless integrated network sensors', *Commun. ACM* **43**(5), 51–58.

Pouzet, M. (2006), *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI.

Prabh, K. S. (2007), Real-Time Wireless Sensor Networks, PhD thesis, The School of Engineering and Applied Science.

Raghunathan, V., Schurgers, C., Park, S. & Srivastava, M. (2002), 'Energy-aware wireless microsensor networks', *Signal Processing Magazine, IEEE* **19**(2), 40–50.

Romberg, J. & Bauer, A. (2004), Loose synchronization of event-triggered networks for distribution of synchronous programs, *in* 'EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software', ACM, New York, NY, USA, pp. 193–202.

Römer, K., Blum, P. & Meier, L. (2005), Time synchronization and calibration in wireless sensor networks, *in* I. Stojmenovic, ed., 'Handbook of Sensor Networks: Algorithms and Architectures', John Wiley & Sons, pp. 199–237.

Romer, K. & Mattern, F. (2004), Event-based systems for detecting real-world states with sensor networks: a critical analysis, *in* 'Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004. Proceedings of the 2004', pp. 389–395.

Rowe, A., Mangharam, R. & Rajkumar, R. (2008), 'Rt-link: A global time-synchronized link protocol for sensor networks', *Ad Hoc Netw.* **6**(8), 1201–1220.

Samper, L., Maraninchi, F., Mounier, L. & Mandel, L. (2006), Glonemo: global and accurate formal models for the analysis of ad-hoc sensor networks, *in* 'InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks', ACM, New York, NY, USA, p. 3.

Schenato, L., Oh, S., Sastry, S. & Bose, P. (2005), Swarm coordination for pursuit evasion games using sensor networks, *in* 'International Conference on Robotics and Automation', pp. 2493–2498.

Sen, S. & Cardell-Oliver, R. (2006), A rule-based language for programming wireless sensor actuator networks using frequency and communnication, *in* 'EMNETS'06'.

Sikka, P., Corke, P., Valencia, P., Crossman, C., Swain, D. & Bishop-Hurley, G. (2006), Wireless adhoc sensor and actuator networks on the farm, *in* 'IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks', ACM, New York, NY, USA, pp. 492–499.

Simon, D., Cifuentes, C., Cleal, D., Daniels, J. & White, D. (2006), Java&#8482; on the bare metal of wireless sensor devices: the squawk java virtual machine, *in* 'VEE '06: Proceedings of the 2nd international conference on Virtual execution environments', ACM, New York, NY, USA, pp. 78–88.

Simon, G., Maróti, M., Lédeczi, A., Balogh, G., Kusy, B., Nádas, A., Pap, G., Sallai, J. & Frampton, K. (2004), Sensor network-based countersniper system, *in* 'SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems', ACM, New York, NY, USA, pp. 1–12.

Sinopoli, B., Sharp, C., Schenato, L., Schaffert, S. & Sastry, S. (2003), 'Distributed control applications within sensor networks', *Proceedings of the IEEE* **91**(8), 1235–1246.

Song, J., Han, S., Mok, A., Chen, D., Lucas, M. & Nixon, M. (2008), Wirelesshart: Applying wireless technology in real-time industrial process control, *in* 'Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE', pp. 377–386.

Srisathapornphat, C., Jaikaeo, C. & Shen, C.-C. (2000), Sensor information networking architecture, *in* 'Parallel Processing, 2000. Proceedings. 2000 International Workshops on', pp. 23–30.

Stankovic, J., Abdelzaher, T., Lu, C., Sha, L. & Hou, J. (2003), 'Real-time communication and coordination in embedded sensor networks', *Proceedings of the IEEE* **91**(7), 1002–1022.

Stipanicev, D. & Marasovic, J. (2003), Networked embedded greenhouse monitoring and control, *in* 'Control Applications, 2003. CCA 2003. Proceedings of 2003 IEEE Conference on', Vol. 2, pp. 1350–1355 vol.2.

Sugihara, R. & Gupta, R. K. (2008), 'Programming models for sensor networks: A survey', *ACM Trans. Sen. Netw.* **4**(2), 1–29.

Tamir, D. E. & Kandel, A. (1995), 'Logic programming and the execution model of prolog', *Information Sciences - Applications* **4**(3), 167–191.

Terfloth, K., Wittenburg, G. & Schiller, J. (2006), Rule-oriented programming for wireless sensor networks, *in* 'International Conference on Distributed Computing in Sensor Networks (DCOSS'06)'.

Texas Instruments (2004), Msp430x12x mixed signal microcontroller (rev. c), Technical report.

Vasconcelos, P. B. & Hammond, K. (2003), Inferring cost equations for recursive, polymorphic and higher-order functional programs, *in* 'IFL', pp. 86–101.

Verdone, R., Dardari, D., Mazzini, G. & Conti, A. (2008), *Wireless Sensor and Actuator Networks: Technologies, Analysis and Design*, Academic Press.

Wada, H., Boonma, P. & Suzuki, J. (2008), 'Macroprogramming spatio-temporal event detection and data collection in wireless sensor networks: An implementation and evaluation study', *Hawaii International Conference on System Sciences* **0**, 499.

Wan, P. & Lemmon, M. (2007), Distributed flow control using embedded sensor-actuator networks for the reduction of combined sewer overflow (cso) events, *in* 'Decision and Control, 2007 46th IEEE Conference on', pp. 1529–1534.

Warner-Allen, G., Johnson, J., Ruiz, M., Lees, J. & Welsh, M. (2005), Monitoring volcanic eruptions with a wireless sensor network, *in* 'Proc. Second European Workshop on Wireless Sensor Networks (EWSN'05)'.

Welsh, M. & Mainland, G. (2004), Programming sensor networks using abstract regions, *in* 'NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation', USENIX Association, Berkeley, CA, USA, pp. 3–3.

Whitehouse, K., Sharp, C., Brewer, E. & Culler, D. (2004), Hood: a neighborhood abstraction for sensor networks, *in* 'MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services', ACM, New York, NY, USA, pp. 99–110.

Whitehouse, K., Zhao, F. & Liu, J. (2008), Semantic streams: A framework for composable semantic interpretation of sensor data, *in* K. Römer, H. Karl & F. Mattern, eds, 'EWSN 2006: Third European Workshop on Wireless Sensor Networks, Zurich, Switzerland', Vol. 3868 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 5–20.

Woo, A., Seth, S., Olson, T., Liu, J. & Zhao, F. (2006), A spreadsheet approach to programming and managing sensor networks, *in* 'IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks', ACM, New York, NY, USA, pp. 424–431.

Yao, Y. & Gehrke, J. (2002), 'The cougar approach to in-network query processing in sensor networks', *SIGMOD Rec.* **31**(3), 9–18.

Zhao, F. & Guibas, L. J. (2004), *Wireless sensor networks : an information processing approach*, The Morgan Kaufmann series in networking., Morgan Kaufmann, Amsterdam ; San Francisco. 2004301905 Feng Zhao, Leonidas J. Guibas. Includes bibliographical references (p. 323-345) and index.

Zhao, Y., Liu, J. & Lee, E. A. (2007), A programming model for time-synchronized distributed real-time systems, *in* 'RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium', IEEE Computer Society, Washington, DC, USA, pp. 259–268.