Diss. ETH Nr. 17397

# A State-Based Programming Model for Wireless Sensor Networks

A dissertation submitted to the
**SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH**
**(ETH ZURICH)**

for the degree of
Doctor of Sciences

presented by
**OLIVER KASTEN**
Dipl.-Informatiker, TU Darmstadt
born July 10, 1969
citizen of Germany

accepted on the recommendation of
Prof. Dr. F. Mattern, examiner
Prof. Dr. L. Thiele, co-examiner

2007

# Abstract

Sensor nodes are small, inexpensive, and programmable devices that combine an autonomous power supply with computing, sensing, and wireless communication capabilities. Networks of sensor nodes can be deployed in the environment at a large scale to unobtrusively monitor phenomena of the real world. Wireless sensor networks are an emerging field of research with many potential applications. So far, however, only few applications have actually been realized. This is in part due to the lack of appropriate programming support, which makes the development of sensor-network applications tedious and error prone. This dissertation contributes a novel programming model and development environment for the efficient, modular, and well structured programming of wireless sensor nodes.

Today there are two principal programming models used for sensor nodes, the *multi-threaded model* and the *event-driven model*. The multi-threaded model requires system support that is often considered too heavy for sensor nodes that operate at the low end of the resource spectrum. To cope with this issue, the event-driven model has been proposed. It requires very little runtime support by the system software and can thus be implemented even on the most constrained sensor nodes.

The simple and lightweight approach to system software, however, tends to make event-driven *applications* in turn quite memory inefficient: Since the event-driven model limits the use of local variables, programmers need to store temporary data in global variables. The memory of global variables, however, cannot easily and automatically be reused, hence the memory inefficiency. To counter this effect, programmers can resort to manual memory management, though this significantly affects program correctness and code modularity. In addition to its drawback of memory inefficiency, event-driven programming requires developers to manually keep track of the current program-state, which makes code modularization and debugging difficult, and leads to unstructured code.

The key contribution of this dissertation is to show that the inadequacies of the event-driven model can be remedied without impairing its positive aspects, particularly its memory-efficient realization in sensor-node system software.

Concretely, we present the Object State Model (OSM), a programming model that extends the event-driven programming paradigm with a notion of hierarchical and concurrent program states. Our thesis is that such *a state-based model allows to specify well-structured, modular, and memory-efficient programs, yet requires as few runtime-resources as the event-driven model.* To support this claim, we also present a programming environment based on the OSM model (including a programming language and compiler), as well as a sensor-node operating system capable of executing OSM programs.

The main idea behind OSM is to explicitly model sensor-node programs as state machines, where variables are associated with states and computational operations are associated with state transitions. In OSM, states serve three purposes. Firstly, states are used as scoping mechanism for variables. The scope and lifetime of variables attached to a state is confined to that state and all of its substates. The memory for storing a state's variables is automatically reclaimed by the runtime system as the program leaves the corresponding state. State variables can be thought of as the local variables of OSM. As such they represent a great advancement over event-driven programming, where the majority of variables effectively have global scope and lifetime. By modeling temporary data with state variables, the use of OSM can significantly increase a program's memory efficiency.

Secondly, the explicit notion of program states allows to model the structure and control flow of a program on a high abstraction level. Specifically, programs can be initially specified in terms of coarse modules (i.e., states), which can be subsequently refined (with substates), leading to modular and readable program code. The third purpose of states is to provide a context for computational operations. States clearly define which variables are visible, and at what point in the control flow the program resides when an operation is executed. In the event-driven model, in contrast, the program's context has to be maintained manually, which typically constitutes a significant fraction of the code, thus making the program hard to read and error prone.

The OSM programming language captures the three concepts described above in order to foster memory efficient, modular, and well-structured programs. A compiler for the proposed language transforms state-based OSM programs back into an event-driven program notation, adding code for automatic memory-management of state variables and code for automatic control-flow management. The compiler-generated code is very lean and does not impose additional requirements on the system software, such as dynamic memory management. Rather, the transformed programs are directly executable on our event-driven system software for resource-constrained sensor nodes. Our language, compiler, and sensor-node system software form the basis of our thesis and constitute a complete state-based programming environment for resource-constrained sensor nodes based on OSM.

# Kurzfassung

Drahtlose Sensorknoten sind kostengünstige, programmierbare und vernetzte Kleinstcomputer, die Sensorik, drahtlose Kommunikation sowie Energieversorgung in sich vereinen. Im grossen Massstab eingesetzt können drahtlose Netze aus solchen Sensorknoten Phänomene der realen Welt unauffällig beobachten. Drahtlose Sensornetze sind ein relativ neues Forschungsfeld mit vielen potentiellen Anwendungen. Unter anderem auf Grund des Fehlens geeigneter Unterstützung bei der Programmierung von Sensorknoten sind von diesen Anwendungen jedoch erst die wenigsten realisiert worden. Diese Dissertation liefert ein neuartiges Programmiermodell mit dazugehöriger Entwicklungsumgebung als Beitrag zur effizienten, modularen und wohlstrukturierten Programmierung von Sensorknoten.

Heutzutage werden zur Programmierung von Drahtlosen Sensorknoten im Wesentlichen zwei Modelle verwendet: das *Multi-Threading Modell* und das *ereignisbasierte Modell*. Das Multi-Threading Modell erfordert Systemunterstützung, die häufig als zu schwergewichtig empfunden wird, insbesondere für Sensorknoten die am unteren Ende des Ressourcenspektrums betrieben werden. Um solchen Ressourcenbeschränkungen gerecht zu werden, wurde das ereignisbasierte Programmiermodell vorgeschlagen. Dieses Modell erfordert sehr geringe Systemunterstützung und kann deshalb auch auf sehr beschränkten Sensoren eingesetzt werden.

Währenddessen jedoch die Systemunterstützung für ereignisbasierte Programme nur wenige Ressourcen erfordert, sind ereignisbasierte Programme selbst sehr speicher*in*effizient. Das ereignisbasierte Modell verhindert nämlich die effiziente Nutzung von lokalen Variablen. Als Konsequenz daraus müssen Programmierer temporäre Daten in globalen Variablen speichern, wodurch die effiziente und automatische Wiederverwendung von Variablenspeicher verhindert wird. Für temporäre Daten könnte zwar der Variablenspeicher auch manuell wiederverwendet werden. Das schwört jedoch Probleme mit der Modularität und der Fehleranfälligkeit herauf. Zusätzlich zum Nachteil der Speicherineffizienz erzwingt das ereignisbasierte Modell ein Programmierparadigma, das Modularisierung und Fehlersuche signifikant erschwert und des Weiteren zu unstrukturiertem Code führt.

Der wesentliche Beitrag dieser Dissertation ist zu zeigen, dass die genannten Unzulänglichkeiten des ereignisbasierten Modells behoben werden können, ohne seine positiven Eigenschaften (also die speichereffiziente Implementierbarkeit auf Sensorknoten) massgeblich zu beeinträchtigen.

Konkret stellen wir das *Object State Model* (OSM) vor, ein Programmiermodell das das ereignisbasierte Modell um die Abstraktion von hierarchisch- und nebenläufig-strukturierbaren Programmzuständen erweitert. Wir vertreten die

These, dass ein solches *zustandsbasiertes Modell die Spezifikation von klar strukturierten, modularen und speichereffizienten Programmen erlaubt und dennoch nicht mehr Ressourcen zu seiner Unterstützung auf Systemebene benötigt als das ereignisbasierte Modell*. Zur Abstützung unserer These präsentieren wir eine Entwicklungsumgebung basierend auf diesem zustandsbasierten Programmiermodell (bestehend aus Programmiersprache und Compiler) sowie ein Betriebssystem zur Ausführung von OSM Programmen.

Die wesentliche Idee hinter OSM ist es Sensorknoten-Programme explizit als Zustandsmaschinen zu modellieren, wobei Variablen mit Zuständen assoziiert werden und Berechnungen mit Zustandsübergängen. Zustände in OSM erfüllen dann drei Aufgaben. Zum einen dienen sie als Geltungsbereich für Variablen. Der Geltungsbereich, und damit die Lebensdauer, jeder Variable ist an genau einen Zustand und dessen Unterzustände gebunden. Wenn das Programm diesen Zustand verlässt, wird der Speicher aller mit dem Zustand assoziierten Variablen freigegeben und kann wiederverwendet werden. Solche an Zustände gebundene Variablen können deshalb als lokale Variablen von OSM betrachtet werden. Sie stellen eine grosse Errungenschaft gegenüber dem ereignisbasierten Modell dar, in dem die Mehrzahl der Variablen einen globalen Geltungsbereich haben und ihr Speicher deshalb nicht wiederverwertet werden kann. Die Spezifikation von temporären Daten mit Zustandsvariablen von OSM kann die Speichereffizienz einer Anwendung signifikant steigern. Als Zweites erlauben es explizite Zustände die Struktur und den Kontrollfluss eines Programmes auf hohem Abstraktionsniveau zu beschreiben. Insbesondere können Programme zuerst grob und mit wenigen Zuständen beschrieben werden, die im Weiteren durch Einfügen von Unterzuständen schrittweise verfeinert werden. Dieses Vorgehen führt zu einem modularem Programmaufbau und gut lesbarem Programm-Code. Zuguterletzt definieren explizite Programmzustände einen Kontext für Berechnungen. Zustände definieren klar an welcher Stelle im Kontrollfluss sich ein Programm befindet, welche Variablen sichtbar sind, welche Zustände bereits durchlaufen und welche Funktion bereits ausgeführt wurden. Im ereignisbasierten Modell dagegen müssen Informationen zum Programmkontext manuell vom Programmierer verwaltet werden. Typischerweise macht diese manuelle Zustandsverwaltung einen grossen Teil des gesamten Programm-Codes aus, was sich in schlechter Lesbarkeit und hoher Fehleranfälligkeit niederschlägt.

Durch die oben beschriebenen Konzepte erleichtert OSM die Beschreibung von speichereffizienten, modularen, wohlstrukturierten und lesbaren Programmen. Der Compiler für die vorgeschlagene Sprache überführt zustandsbasierte OSM-Programme zurück in eine ereignisbasierte Repräsentation. Dabei wird Code zur automatischen Speicherverwaltung von Variablen und zur Verwaltung des Kontrollflusses hinzugefügt. Der vom Compiler erzeugte Code ist leichtgewichtig und erfordert keine weitere Unterstützung auf Systemebene, wie z.B. dynamisch Speicherverwaltung. Vielmehr sind die erzeugten Programme direkt auf unserem ereignisbasiertem Betriebssystem für ressourcenbeschränkte Sensorknoten ausführbar. Die von uns entwickelte vollständige Programmierumgebung für Sensorknoten, bestehend aus Programmiersprache, Compiler, und Betriebssystem, zeigt dies exemplarisch und bildet somit die Grundlage für diese Dissertation.

# Contents

# 1 Introduction

## 1.1 Background

Over the past decades, advances in sensor technology, miniaturization and low-cost, low-power chip design have led to the development of wireless sensor nodes: small, inexpensive, and programmable devices that combine computing, sensing, short-range wireless communication, and an autonomous power supply. This development has inspired the vision of wireless sensor networks—wireless networks of sensor nodes that can be deployed in the environment at the large scale to unobtrusively monitor phenomena of the real world. There is a wide range of applications envisioned for such wireless sensor networks, including precision agriculture, monitoring of fragile goods and building structures, monitoring the habitat of endangered species and the animals themselves, emergency response, environmental and micro-climate monitoring, logistics, and military surveillance.

The most popular programming model for developing sensor-network applications today is the event-driven model. Several programming frameworks based on that model exist, for example, the BTnode system software [21], Contiki [37], the TinyOS system software [59] and nesC programming language [45], SOS [52] and Maté [79]. The event-driven model is based on two main abstractions: events and actions. Events in the context of event-driven programming can be considered abstractions of real-world events. Events are typically typed to distinguish different event classes and may contain parameters, for example, to carry associated event data. At runtime, the occurrence of an event can trigger the execution of a computational action. Typically, there is a one-to-one association between event types and actions. Then, an event-driven program consists of as many actions as there are event types. Actions are typically implemented as functions of a sequential programming language. They always run to completion without being interrupted by other actions. In order not to monopolize the CPU for any significant time, actions need to be non-blocking. Therefore, at any point in the control flow where an operation needs to wait for some event to occur (e.g., a reply message or acknowledgment), the operation must be split into two parts: a non-blocking operation request and an asynchronous completion event. The completion event then triggers another action, which continues the operation.

One of the main strengths of the event-driven model is that its support in the underlying system software incurs very little memory and runtime overhead. Also, the event-driven model is simple, yet intuitively captures the reactive nature of sensor-network applications. In contrast, the multi-threaded programming model, another programming approach that has been applied to wireless sensor networks, is typically considered to incur too much memory and computational overhead (mainly because of per-thread stacks and context switches).

This makes the multi-threaded model generally less suitable for programming sensor nodes, which are typically limited in resources, and unsuitable for sensor nodes operating at the low end of the resource spectrum (cf. [37, 59]). As a consequence, in the context of sensor networks, event-based systems are very popular with several programming frameworks in existence. These frameworks, particularly TinyOS and nesC, are supported by a large user base. Hence, many sensor-node programmers find it natural to structure their programs according to this paradigm.

## 1.2 Motivation and Problem Statement

While event-driven programming is both simple and efficient, we have found that it suffers from a very important limitation: it lacks an abstraction of state. Very often, the behavior of a sensor-node program not only depends on a particular event—as the event model suggests—but also on the event history and on computational results of previous actions; in other words: program state. We use an example to illustrate this point. In the EnviroTrack [6] middleware for tracking mobile objects with wireless sensor networks, nodes collaboratively track a mobile target by forming a group of spatially co-located nodes around the target. If a node initially detects a target, its reaction to this target-detection event depends on previous events: If the node has previously received a notification (i.e., an event) that a nearby node is already a member in a tracking group, then the node joins that group. If the node has not received such a notification, it forms a new group. In this example, the behavior of a node in reaction to the target-detection event (i.e., creating a new tracking group or joining an existing one) depends on both the detection event itself and the node's current state (i.e., whether a group has already been formed in the vicinity of the node or not).

Since there is no dedicated abstraction to model program state in the event-driven model, programmers typically save the program state in variables. However, keeping state in variables has two important implications that obscure the structure of program code, severely hamper its modularity, and lead to issues with resource efficiency and correctness. Firstly, to share state across multiple actions, the state-keeping variables must persist over the duration of all those actions. Automatic (i.e., local) variables of procedural programming languages do not meet this requirement, since variable lifetime is bound to functions. Thus the local variable stack is unrolled after the execution of the function that implements an action. Instead, programmers have the choice to use either global variables or to manually store a state structure on the heap. The latter approach is often inapplicable as it requires dynamic memory management, which is rarely found in memory-constrained embedded systems because it is considered error-prone and because it incurs significant memory overhead. The former and by far more typical approach, however, also incurs significant memory overhead. Since global variables also have "global" lifetime, this approach permanently locks up memory—even if the state is used only temporarily, for example, during network initialization. As a result, manual state-keeping makes event-driven programs memory inefficient since they cannot easily reuse memory for temporary data. This is ironic, since one of the main reasons for introducing the

event-driven model has been that its support in the underlying system software can be implementred resource efficiently.

Secondly, the association of events to actions is static—there is no explicit support for adopting this association depending on the program state. As a consequence, in actions, programmers must manually dispatch the control flow to the appropriate function based on the current state. The additional code for state management and state-based function demultiplexing can obscure the logical structure of the application and is an additional source of error. Also, it hampers modularity since even minor changes in the state space of a program may require modifying multiple actions and may thus affect much of the program's code base.

While the identified issues are not particularly troublesome in relatively small programs, such as application prototypes and test cases, they pose significant problems in larger programs. But as the field of wireless sensor networks matures, its applications are getting more complex—projects grow and so does the code base and the number of programmers involved. Leaving these issues unsolved may severely hamper the field of wireless sensor networks to mature.

## 1.3 Thesis Statement

Because of the limitations outlined in the previous section, we argue that the simple event/action-abstraction of the event-driven programming model is inadequate for large or complex sensor network programs. The natural question then is, can the inadequacies of the event-driven programming model be repaired without impairing its positive aspects? In this dissertation we answer this question affirmatively. Concretely, we present the Object State Model (OSM), a programming model that extends the event-driven programming paradigm with an explicit abstraction and notion of hierarchical and concurrent program states. Our thesis is that such *a state-based model allows to specify well-structured, modular, and memory-efficient sensor-node programs, yet requires as little runtime-resources as the event-driven model*.

## 1.4 Contributions

The main contribution of this work then is to show

- that OSM provides adequate abstractions for the specification of well structured and highly modular programs,
- how OSM supports memory-efficient programming, and, finally,
- that OSM does not incur significant overhead in the underlying system software and can thus indeed be implemented on resource-constrained sensor nodes.

In this dissertation we present four elements to support our claim: (1) OSM, an abstract, state-based programming model, which is a state-extension to the conventional event-based programming model, (2) an implementation language

for sensor nodes that is based on the OSM model, (3) a compiler for the proposed language that generates event-driven program code, and (4) an execution environment for the generated code on BTNODE sensor-node prototypes. The following paragraphs present the key concepts of our solution, namely using an explicit notion of state for structuring programs, automated state management, efficient memory management based on the use of state as a lifetime qualifier for variables, and the automatic transformation of OSM programs into conventional event-driven programs.

### 1.4.1 Modular and Well-Structured Design

In the state-based model of OSM, the main structuring element of program code is an explicit notion of program state. For every explicit program state the programmer can specify any number of actions together with their trigger conditions (over events). At runtime, actions are only invoked when their associated program state is assumed and when their trigger condition evaluates to true. As a consequence, the association of events to actions in OSM depends on the current state (unlike in the event-driven model), that is, invocations of actions become a function of both the event and the current program machine state. This concept greatly enhances modularity, since every state and its associated actions can be specified separately, independent of other states and their actions. Thus, modifying the state space of a program requires only local changes to the code base, namely to the implementations of the affected states. In contrast, in the event-driven model, such modifications typically affect several actions throughout much of the entire code.

### 1.4.2 Automated State Management

Generally, OSM moves program-state management to a higher abstraction layer and makes it an explicit part of the programming model. Introducing state into the programming model allows to largely automate the management of state and state transitions—tasks which had to be performed manually before. The elimination of manually written code for the management of state-keeping variables and state-based function demultiplexing allows programmers to focus on the logical structure of the application and removes a typical source of error.

### 1.4.3 Memory-Efficient State Variables

Even in state-based models, variables can be a convenient means for keeping program state, for example, to model a very large state space that would be impractical to specify explicitly or to store state that does not directly affect the node's behavior. Instead of having to revert to wasteful *global variables* (as in the event-driven model), OSM supports variables that can be attributed to an explicitly modeled state. This state can then be utilized to provide a novel scope and lifetime qualifier for these variables, which we call *state variables*. State variables exist and can share information—even over multiple action invocations—when their associated state is assumed at runtime. During compilation, state variables of mutually exclusive states are automatically mapped to overlapping

memory regions. The memory used to store state variables can be managed automatically by the compiler, very much like automatic variables in procedural languages. However, this mapping is purely static and thus requires neither a runtime stack nor dynamic memory management.

State variables can reduce or even eliminate the need to use global variables or manually memory-managed variables for modeling temporary program state. Since state variables provide automatic memory management, they are not only more convenient to use but also more memory efficient and less error-prone. Because of state variables OSM programs are generally more memory efficient compared to event-driven programs.

### 1.4.4 Light-Weight Execution Environment

The state-based control structure of OSM programs can be automatically compiled into conventional event-driven programs. Generally, the automatic transformation, which is implemented by the OSM compiler, works like manually coding state in event-driven systems. In the transformation, however, the program-state space is automatically reduced and then converted into a memory-efficient representation as variables of the event program. State-specific actions of OSM are transformed into functions of a host language. Then, code for dispatching the control flow to the generated functions is added to the actions of the event program. After the transformation into event-driven code, OSM programs can run in the same efficient execution environments as conventional event-driven systems. Thus, the state-based model does not incur more overhead than the (very efficient) event-driven model and requires only minimal operating system support.

### 1.4.5 Evaluation

We use the state-based OSM language to show that a first class abstraction of state can indeed support well-structured and modular program specifications. We also show that an explicit abstraction of state can support memory-efficient programming. To do so, we present an implementation of state variables for the OSM compiler. More concretely, we present an algorithm deployed by our OSM compiler that automatically reclaims unused memory of state variables. Finally, we show that code generated from OSM specifications can run in the very light-weight event-driven execution environment on BTnodes.

## 1.5 Thesis Organization

The remainder of this dissertation is structured as follows. In Chapter 2 we discuss general aspects of wireless sensor networks. First we characterize wireless sensor networks and present envisioned as well as existing applications. Then we discuss state-of-the-art sensor-node hardware, ranging from early research prototypes to commercialized devices.

In Chapter 3 we present a general background on programming individual sensor nodes. We start by discussing some general requirements of sensor-node

programming. Then we present the three dominant programming models in use today. We will particularly focus on the runtime support necessitated by those models and evaluate their suitability with respect to the requirements established previously. Finally, we present selected state-of-the-art sensor-node programming frameworks. Chapter 4 is devoted to the detailed examination of the event-driven programming model. Based on our experience with the event-driven BTnode system software and other programming frameworks described in the current literature, we analyze the model's limitations and drawbacks. We lay down why the convential event-driven model is not well-suited to describe sensor-node applications and we stress how a first-class abstraction of program states could indeed solve some of the most pressing issues.

The next three chapters are devoted to our state-based programming framework OSM. Chapter 5 presents the OSM model in detail. In particular, the semantics of the model are discussed. The OSM language and implementation aspects of the OSM compiler are examined in Chapter 6. We present the OSM specification language and its mapping to the event-driven execution environment on BTnodes. The execution environment of OSM programs on BTnodes is a slightly modified version of our event-driven system software presented in Sect. 3.5.1. In Chapter 7 we evaluate the practical value of OSM. First we discuss how OSM meets the requirements for programming individual sensor nodes as set out in previous chapters. Then we discuss how OSM alleviates the identified limitations of the event-based model. Finally, we show how OSM can be used to implement concrete applications from the sensor network literature.

In Chapter 8 we present related work. Chapter 9 concludes this dissertation, discusses the limitations of our work, and ends with directions for future work.

# 2 Wireless Sensor Networks

The goal of this dissertation is to develop a software-development framework appropriate for programming wireless sensor network (WSN) applications. To achieve this goal, we first need to understand what makes one programming technique appropriate and another one inadequate. Software, as well as the techniques for programming this software, are always shaped by its applications and by the computing hardware. So in order to develop an understanding for the characteristics, requirements, and constraints of WSNs, we will present and analyze a range of current and envisioned WSN applications and their computational platforms, the sensor nodes.

WSNs have a wide range of applications. The exact characteristics of a particular sensor network depend largely on its application. However, some characteristics are shared by most, if not all WSNs. The most important characteristics are: (1) WSNs monitor physical phenomena by sampling sensors of individual nodes. (2) They are deployed close to the phenomenon and (3), once deployed, operate without human intervention. And finally (4), the individual sensor nodes communicate wirelessly. These general characteristics together with the application-specific characteristics translate more or less directly into technical requirements of the employed hardware and software and thus shape (or at least should shape) their design. Indeed, the technical requirements of many of the more visionary WSN applications go far beyond what is technically feasible today. To advance the field of WSNs, innovative hardware and software solutions need to be found.

Regarding WSN software, classical application architectures and algorithm designs need to be adapted to the requirements of WSNs. However, this is not enough. Today's programming models and operating systems also do not match WSN requirements and are thus equally ill-suited to support such new software designs. Programming models must be revised in order to facilitate the design of WSN applications with expressive abstractions. And operating systems need to be adapted to support these new programming models on innovative sensor-node hardware designs.

Before looking at the software development tools and process in the next chapter, we present the characteristics of WSN applications and their computational platforms in this chapter. We start by presenting a number of application scenarios in Sect. 2.1. From these applications we derive common application characteristics in Sect. 2.2. Some of the software requirements can be derived directly from those characteristics, while others are imposed by the characteristics of sensor nodes. Therefore we present classes of sensor-node designs, their hardware components, and selected sensor-node hardware platforms in Sect. 2.3. We explain how these hardware designs try to meet the WSN requirements and which constraints they impose on the software themselves. Before summarizing this chapter in Sect. 2.5 we briefly relate wireless sensor nodes to embedded systems

in Sect. 2.4.

## 2.1 WSN Applications

WSN share a few common characteristics, which make them recognizable as such. However, sensor networks have such a broad range of existing and potential applications, that these few common characteristics are not sufficient to describe the field comprehensively. In this section we present concrete WSN applications to exemplify their diversity. Based on this presentation we derive WSN characteristics in the next section. Similar surveys on the characteristics of WSNs can be found in [70, 101].

### 2.1.1 Environmental Observation

One application domain of WSN is the observation of physical phenomena and processes in a confined geographical region, which is covered by the WSN. Several WSNs have already been deployed in this domain. For example, a WSN has been used to monitor the behavior of rocks beneath a glacier in Norway [88]. The goal of the deployment is to gain an understanding for the sub-glacier dynamics and to estimate glacier motion. Sensor nodes are placed in drill holes in and beneath the glacier. The holes are cut with a high-pressure hot water drill. The sensor nodes are not recoverable; their width is constrained by the size of the drill hole, which is 68 mm. Communication between the nodes in, and the base station on top of the glacier follows a pre-determined schedule. To penetrate the up to 100 meters of ice, the nodes boost the radio signal with an integrated 100 mW power amplifier. The network is expected to operate for at least one year.

To monitor the impact of wind farms on the sedimentation process on the ocean floor and its influence on tidal activity [87] a WSN is being used off the coast of England. The sensor nodes are dropped from ships at selected positions and are held in position by an anchor. Each of the submerged sensors is connected to a floating buoy, which contains the radio transceiver and GPS receiver. The nodes measure pressure, temperature, conductivity, current, and turbidity.

Another maritime WSN deployment called ARGO is used to observe the upper ocean [63]. ARGO provides a quantitative description of the changing state of the upper ocean and the patterns of ocean climate variability over long time periods, including heat and freshwater storage and transport. ARGO uses free-drifting nodes that operate at depths up to 2000 m below sea-level but surface every 10 days to communicate their measurements via satellite. Each node is about 1.3 m long, about 20 cm in diameter, and weigh about 40 kg. Much of the node's volume is taken up by the pressure case and the mechanism for submerging and surfacing. The per-node cost is about $ 15.000 US; it's yearly operation (i.e., deployment, data analysis, project management, etc.) accounts for about the same amount. The nodes are designed for about 140 submerge / resurface cycles and are expected to last almost 4 years. After that time most nodes will sink to the ocean floor and are thus not recoverable. Currently almost 3000 nodes are deployed.

WSNs are also used in precision agriculture to monitor the environmental factors that influence plant growth, such as temperature, humidity light, and soil moisture. A WSN has been deployed in a vineyard [15] to allow precise irrigation, fertilization, and pest control of individual plants. Other examples for environmental observations are the detection and tracking of wildfires, oil spills, and pollutants in the ground, in water, and in the air.

### 2.1.2 Wildlife and Farm-Animal Monitoring

Monitoring the behavior and health of animals is another application domain of WSNs. Some wild animals are easily disturbed by human observers, such as the Leach's Storm Petrel. To unobtrusively monitor the breeding behavior of these birds in their natural habitat, a WSN has been deployed on Great Duck Island, Maine, USA [85, 112]. Before the start of the breeding season, sensor nodes are placed in the birds' nesting burrows and on the ground. They monitor the burrows' micro climate and occupancy state. About one hundred nodes have been deployed. They are expected to last for an entire breeding period (about seven months), after which they can be recollected. Another WSN described in [117, 118] aims to recognize a specific type of animal based on its call and then to locate the calling animal in real-time. The heterogeneous, two-tired sensor network is composed of so-called micro nodes and macro nodes. The smaller, less expensive, but also less capable micro nodes are deployed in great numbers to exploit spatial diversity. The fewer but more powerful macro nodes combine and process the micro node sensing data. The target recognition and location task is divided into two steps. All nodes first independently determine whether their acoustic signals are of the specified type of animal. Then, macro nodes fuse all individual decisions into a more reliable system-level decision using distributed detection algorithms. In a similar approach, [64, 105] investigates monitoring of amphibian populations in the monsoonal woodlands of northern Australia.

WSNs have also been used or are planned to be used to monitor and track wild species and farm animals, such as cows, deer, zebras, fishes, sharks, or whales. In these scenarios, sensor nodes are attached to individual animals in order to determine their health, location, and interaction patterns. In these applications, WSNs are superior to human observers because often humans cannot easily follow the targeted animals and because human observation is too cost intensive. The ZebraNet [67, 81] WSN is used to observe wild zebras in a spacious habitat at the Mpala Research Center in Kenya. A main research goal is to understand the impact of human development on the species. The WSN monitors the behavior of individual animals (e.g., their activity and movement patterns) and interactions within a species. The nodes, which are worn as a collar, log the sensory data and exchange them with other nodes, as soon as they get into communication range. The data is then retrieved by researchers who regularly pass through the observation area with a mobile base stations mounted on a car or plane.

A restoration project after an oil spill in the Gulf of Alaska identified critical habitats of the Pacific halibut by determining its life-history patterns [102]. The knowledge of the fishes' critical habitat aids fisheries managers in making de-

cisions regarding commercial and sport fishing regulations. The sensor nodes are attached to a number of individual fishes in the form of so-called pop-up tags. The tags collect relevant sensory data and store them. After the observation period a mechanism releases the tags, which then "pop up" to the surface and communicate their data by satellite. Typically tags are lost unless washed ashore and returned by an incidental finder.

WSNs are also used in agriculture, for example in livestock breeding and cattle herding. The Hogthrob project [24], for example, aims at detecting the heat periods of sows in order to determine the right moment of artificial fertilization. The intended area of operation are farms with a high volume of animals (2000 individuals and more). To be economically sensible, sensor nodes must last for 2 years at a cost of about one Euro.

To reduce the overhead of installing fences and improving the usage of feeding lots in cattle herding, a WSN implements virtual fences [28]. An acoustic stimulus scares the cows away from the virtual fence lines. The movement pattern of the animals in the herd is analyzed and used to control the virtual fences. The sensor nodes are embedded into a collar worn by the cow.

### 2.1.3 Intelligent Environments

Applications for WSN cannot only be found in the outdoors. Instead of attaching sensor nodes to animals, they may also be embedded within (or attached to) objects of the daily life. Together with a typically fixed infrastructure within buildings, such WSNs create so-called intelligent environments. In such environments, augmented everyday objects monitor their usage pattern and support their users in their task through the background infrastructure.

The Mediacup [16], a coffee cup augmented with a sensor node, detects the cup's current usage pattern, for example, if it is sitting in the cupboard, filled with hot coffee, drunken from, carried around, or played with. The cup knows when the coffee in the cup is too hot to drink and warns the user. A network of Mediacups can detect whether there is a meeting taking place and automatically adjust the label of an electronic doorplate. Similarly, to support the assembly of complex kits of all kinds, for example, machinery, the individual parts and tools could be augmented with sensor nodes. In a prototypical project described in [12], a WSN assists in assembling do-it-yourself furniture to save the users from having to study complex assembly instructions.

### 2.1.4 Facility Management

To get indications for potential power savings, a WSN monitors the power consumption in office buildings and in private households [69]. Data is measured by the power-adapter like sensors that are located either in electrical outlets or at sub-distribution nodes, such as fuseboxes. Measurements are collected by a central server and can be accessed and interpreted in real-time. Sensor nodes are powered through the power grid in the order of tens to hundreds. Their lifetime should be equivalent to the lifetime of the building. Obviously, their combined power consumption should be very low, that is, well below the savings gained through its employment. Another possible application in the living and

working space are micro-climate monitoring to implement fine-grained control of Humidity, Ventilation and Air Conditioning (HVAC).

### 2.1.5 Logistics and Asset Tracking

Another large application domain is tracking of products and assets, such as cargo containers, rail cars, and trailers. Again, sensor nodes in this domain are attached to the items to be monitored and typically log the items' position and state and notify users of critical conditions.

For example, the AXtracker [82] is a commercial wireless sensor node for asset tracking and fleet management which can report conditions such as a door being open, a decrease or increase in temperature or moisture, and it can detect smoke. SECURIfood [96] is a commercial WSN for cold-chain management. The networks detects inadequate storage conditions of frozen or refrigerated products early in order to issue warnings. In a research project [106] a WSN has been used to augment products as small as an egg carton in a supermarket in order to facilitate the early removal of damaged or spoiled food items. If the product is damaged (e.g., because it has been dropped) or has passed its best-before date, it connects to the PDA or mobile phone of an employee on the sales floor to report its condition. Such a solution is superior to RFID or manual barcode scanning, because sensor nodes can take the individual product and storage history into account. Obviously, sensor nodes need to be very inexpensive, that is, in the range of noticeably less than a few percent of the tagged product, which is the profit margin for food items. The node's life must span the period from production to sale. After consumption of the product the sensor node is disposed with its packaging.

### 2.1.6 Military

Many of the early, mostly US-based WSN research projects were in the military domain. For example, WSNs are used for tracking enemy vehicles [6]. Sensor nodes could be deployed from unmanned aerial vehicles in order to detect the proximity of enemy tanks with magnetometers. In such an application sensor nodes need to be unnoticeably small in order to prevent their detection and removal or the nodes must be too numerous to be removed effectively. In either case, the nodes are lost after deployment. A WSN integrated into anti-tank mines ensures that a particular area is covered with mines [90]. The mines estimate their orientation and position. Should they detect a breach in the coverage, a single mine is selected to fill in the gap. The mine can hop into its new position by firing one of its eight integrated rocket thrusters. Obviously, sensor nodes on mines are disposable. WSNs can also be used in urban areas to locate a shooter. A system has been implemented that locates the shooter by detecting and analyzing the the muzzle blast and shock wave using acoustic sensors [109].

## 2.2 WSN Characteristics

### 2.2.1 Deployment and Environmental Integration

A common characteristic of wireless sensor networks is their deployment in the physical environment. In order to monitor real-world phenomena in situ, sensor nodes are deployed locally, close to where the phenomena is expected. The sensory range and accuracy of individual nodes is typically limited. If applications require more accurate data (e.g., in the shooter localization application) or a complete coverage of an extensive area (e.g., vehicle tracking), nodes must be deployed in larger numbers. Current deployments typically consist of a few to several tenths of sensor nodes, though deployments of several thousand devices are envisioned by researchers.

The way sensor nodes are deployed in the environment differs significantly depending on the application. Nodes may be carefully placed and arranged at selected locations as to yield best sensing results. Or they may be deployed by air-drop resulting in a locally concentrated but random arrangement. Sensor nodes may also be attached directly to possibly mobile objects that are to be monitored, for example, to animals, vehicles, or containers.

For outdoor applications, sensor nodes need to be sealed from environmental influences (such as moisture, fungus, dust, and corrosion) to protect the sensitive electronics. Sealing can be done with conformal coating or packaging. Both strategies may have influences on the quality of sensor readings and wireless communications. Packaging may also contribute significantly to the weight and size of sensor nodes.

### 2.2.2 Size, Weight, and Cost

The allowable size, weight, and cost of a sensor node largely depends on application requirements, for example, the intended size of deployment. Particularly in large scale, disposable (i.e., non recoverable) sensor node deployments, individual nodes must be as cheap as possible. In the military domain, for example, Smart Dust sensor nodes [120] aim at a very large scale deployment of nodes the size of one cubic millimeter and very low cost. For commercial applications (such as logistics, product monitoring, agriculture and animal farming, facility management, etc.), cost will always be a prime issue. On the other extreme, nodes can be as big as suitcases and cost up to several hundred Euros, for example, for weather and ocean monitoring. Application scenarios in which the sensor network is to float in the air pose stringent weight restrictions on nodes.

We expect that future sensor-node technology will typically meet the weight and size requirements, but that stringent requirements for low cost will significantly limit the nodes' energy budget and available resources (see below).

### 2.2.3 Limited Energy Budget

Wireless sensor nodes typically have to rely on the finite energy reserves from a battery. Remote, untethered, unattended, and unobtrusive operation of sensor networks as well as the sheer number of nodes in a network typically precludes

the replacement of depleted batteries. Harvesting energy from the environment is a promising approach to power sensor nodes currently under research. First results suggest that under optimal environmental conditions, energy harvesting can significantly contribute to a node's energy budget, but may not suffice as sole energy resource [99].

Each WSN application has specific constraints regarding size, weight, and cost of individual sensor nodes. These factors directly constrain what can be reasonably integrated into sensor nodes. Particularly, power supplies contribute significantly to the size, weight, and cost of sensor nodes. In the majority of WSN applications, energy is a highly limited resource.

### 2.2.4 Lifetime

The application requirements regarding sensor network lifetime can vary greatly from a few hours (e.g., furniture assembly) to many years (e.g., ocean monitoring). But also the notion of WSN lifetime can vary from application to application. While some WSN applications can tolerate a large number of node failures and still produce acceptable results with only a few nodes running, other applications may require dense arrangements. But the network lifetime will always depend to a large degree on the lifetime of individual sensor nodes.

Sensor-node lifetime mainly depends on the node's power supply and its power consumption. But as the available energy budget is often tightly constrained by application requirements on the node's size, wight and cost and, at the same time, lifetime requirements are high, low-power hardware designs and energy aware software algorithms are imperative. Today, sensor network lifetime is one of the major challenges of WSN research.

### 2.2.5 Limited Computing Resources

Sensor nodes designs often trade off computing resources for one of the fundamental characteristics of sensor nodes, size, monetary cost, and lifetime (i.e., energy consumption). Computing resources, such as sensors, wireless communication subsystems, processors, and memories, significantly contribute to all three of them. Sensor-node designs often try to strike a balance between sensor resolution, wireless communication bandwidth and range, CPU speeds, and memory sizes on the one hand, and cost, size, and power consumption on the other hand.

For many WSN applications, cost is a crucial factor. Assuming mass production, the price of integrated electronics and thus the price per sensor node is mainly a function of the die size, that is, the number of transistors integrated. The size of the node on the die determines the how many devices can be produced from a single wafer while wafers induce about constant costs regardless of what is produced from them. That is the main reason why complex 32-bit microcontroller architectures are much more expensive then comparatively simple 8-bit architectures. As we will detail in the next section, where we will look at selected sensor-node components, an increase of the address-bus size from 8 to 32-bit of typical sensor-node processors today leads to a ten-fold increase in processor price. Moore's law (see [91]) predicts that more and more transistors

can be integrated, and thus integrated electronics get smaller and less expensive in the future. While this means that sensor-network applications with more and more (possibly disposable) nodes can be deployed, it does not necessarily imply that the resources of individual nodes will be less constrained. Memories in general, and SRAM in particular, require high transistor densities, and thus are also a significant cost factor.

Also, the energy consumption of sensor nodes mainly depends on the deployed hardware resources (and their duty cycle). More powerful devices quite literally consume more power. Most of the power consumption of integrated devices stems from switching transistors (a CPU cycle, refreshing memories, etc). Again, less powerful processor architectures with less gates have an advantage.

As a consequence, computing resources are often severely constrained, even when compared to traditional embedded systems, which have similar cost constraints but are typically mains powered. We expect that these limitations of sensor networks will not generally change in the future. Though we can expect advances in chip technology and low-power design, which will lift the most stringent restrictions of current applications. But the attained reductions in per-node price, size, and energy consumption also open up new applications with yet tighter requirements. Less powerful devices will remain to have the edge in terms of price and power consumption.

### 2.2.6 Collaboration

The collective behavior of a wireless sensor network as a whole emerges from the cooperation and collaboration of the many individual nodes. Collaboration in the network can overcome and compensate for the limitations of individual nodes. The wireless sensor network as a whole is able to perform tasks that individual nodes cannot and that would be difficult to realize with traditional sensing systems, such as satellite systems or wired infrastructures.

### 2.2.7 Back-End Connectivity

To make the monitoring results of WSN available to human observers or control applications, they may be connected to a fixed communication infrastructure, such as Wireless LAN, satellite networks, or GSM networks. The WSN may be connected to this communication infrastructure via one or several gateways. Connectivity to a communication infrastructure requires that either nodes from the WSN operate as gateways, or that fixed gateways are installed. Such gateway typically posses two network interfaces. Acting as a gateway between the WSN and the communication infrastructure (which is typically not optimized for low power consumption) places additional burdens on sensor nodes. On the other hand, installing fixed gateways is often too expensive.

Rather than relaying data from the WSN into a fixed infrastructure, monitoring results may also be retrieved by mobile terminals. In ZebraNet, for example, sensing results are collected by humans with laptop computers [67]. Since permanent connectivity from the WSN to the mobile terminals cannot be guaranteed, sensor nodes are required to temporarily store their observations.

### 2.2.8 Mobility and Network Dynamics

Sensor nodes may change their location after deployment. They may have automotive capabilities, as described in the military application above or they may be attached to mobile entities, like animals. Finally, nodes may be carried away by air and water currents.

If nodes are mobile, the communication topology may change as nodes move out of communication range of their former communication partners. However, the network topology may change even if nodes are stationary, because nodes fail or more nodes are deployed. Additionally, communication links may be temporarily disrupted as mobile objects, such as animals or vehicles, or weather phenomena obstruct RF propagation. Finally, some nodes may choose to turn their radio off in order to save power. If the sensor network is sparsely connected, individual node or link failures may also result in network partitions.

As a consequence of their integration into the environment sensor nodes may be destroyed by environmental influences, for example, corroded by sea water or crushed by a glacier. If batteries cannot be replaced, they may run out of power. Increasing the node population (and hence the quality and quantity or data readings) may be desirable at places that have been found to be interesting after the initial deployment. After an initial deployment, a (possibly repeated) re-deployment of sensor nodes may become necessary in order to replace failed nodes or to increase the node population at a specific location. All these factors may lead to a dynamic network, with changing topologies, intermittent partitions, and variable quality of service.

### 2.2.9 Bursty Traffic

Wireless sensor networks may alternate between phases of low-datarate traffic and phases of very bursty, high-datarate traffic. This may be the case, for example, if the wireless sensor network is tasked to monitor the occurrence of certain phenomena (as opposed to continuous sampling), and that phenomenon is detected by a number of nodes simultaneously.

### 2.2.10 Dynamic Role Assignment

To optimize certain features of the sensor network, such as network performance, lifetime, and sensing quality, the nodes of a sensor network may be dynamically configured to perform specific functions in the network. This so-called *role assignment* [42, 98] is based on static as well as dynamic parameters of the nodes, such as hardware configuration, network neighbors, physical location within the network (e.g., edge node or distance to the infrastructure gateway), or remaining battery levels. For example, in dense networks, where all areas of interest are covered by multiple nodes (both in terms of network and sensor coverage), some of the nodes may be switched off temporarily to conserve energy. These nodes can then later replace the nodes that have run out of battery power in the meantime, thus increasing the overall lifetime of the sensor network [122]. As it is often difficult to anticipate the node parameters before deployment, role assignment is typically performed in situ by the sensor network itself. Typically,

it is first performed right after deployment, in the network initialization phase. Then, changing parameters of a node and its environment may regularly prompt reassignment of the node's role.

### 2.2.11 Node Heterogeneity

While many sensor networks consist of identical nodes, for some applications it can be advantageous to deploy multiple hardware configurations. For example, nodes may be equipped with different sensors; some nodes may be equipped with actuators, while others may be equipped with more computing power and memory. Nodes with a powerful hardware configuration typically consume more energy, but in turn can run more complex software and perform special functions in the network, such as communication backbone, gateway to the background infrastructure, or host for sophisticated computations. Low capability nodes typically perform simpler tasks but have higher lifetimes at lower cost. These setups can lead to clustered and multi-tiered architectures, where the cluster-heads are equipped with more computing and memory resources.

## 2.3 Sensor Nodes

As we have seen in the previous sections, different applications have different requirements regarding lifetime, size, cost, etc. While functional prototypes have been realized for most the applications described above, many of the employed sensor-nodes do not meet all their requirements. Concretely, most devices are either too big, too expensive for the application at hand, or they fall short of lifetime requirements. Therefore miniaturization, low-power and low-cost designs are possibly the most pressing technical issues for sensor nodes.

Most sensor nodes have been built as research prototypes or proof-of-concept implementations used to investigate certain aspects of future sensor nodes, networks, algorithms, and applications. Indeed, even the designs which have been commercialized are targeted at research institutions and very early adopters.

### 2.3.1 Device Classes

In today's WSN deployments and research installations we see three classes of sensor nodes of different degrees of maturity. The first class are commodity devices that are being used as sensor nodes, though they are not actually built for that purpose. Devices in that class are laptop computers, PDAs, mobile phones, cameras, etc. Commodity devices are often used for rapid application prototyping, where the actual device characteristics are less relevant. The second class of sensor nodes are custom built from commercially-available electronics components. These nodes are closer to the actual requirements of WSN and often perform better than commodity devices. However, they still do not meet most of the envisioned characteristics, such as "mote" size, years of lifetime, and a price in the order of cents. Typically these nodes target at a broad applicability. Some nodes designs are available commercially, such as the latest version of the BTnode. Because of their availability and relatively low cost, these nodes

dominate current WSN deployments. The last class of sensor nodes, finally, are highly integrated designs that combine all the functional parts of a sensor node (CPU, memories, wireless communication, etc.) into a single chip or package. These devices are the best "performers" but still remain to be research prototypes produced in very few numbers only. Below we will discuss these classes of sensor node in more detail.

### Commodity Devices

Rather than building custom sensor nodes, some researchers have utilized commercially available commodity devices to build prototypical sensor network algorithms and applications. Commodity devices, such as laptop computers, PDAs, mobile phones, cameras, etc., can be used to perform special functions in the network. In some research projects they even have been operated as sensor node replacements.

There are a number of benefits to this approach: commodity devices are readily available and provide a variety of sensors and resources. Many commodity devices provide standardized wired and wireless interfaces (such as RS-232 serial ports, USB, Bluetooth, and Infrared) and application protocols (e.g, RFCOM and OBEX), which allow to use the device's functionality without a major programming effort. For laptop computers, for example, there is an immense variety of peripherals available. Custom-made peripherals can be connected to the computer via its general-purpose IO interface. When programming is required, software development on commodity devices is typically more convenient compared to custom-made sensor nodes because these devices offer established software-development environments and operating-system support. Furthermore, no knowledge of embedded-system design and chip design is required, thereby opening up the possibility of work in WSNS even for non-experts in embedded-systems design.

There are several examples where commodity devices are used in WSN. In [117, 118], for example, PDAs and embedded PCs, respectively, are used as cluster heads. For wirelessly interfacing the sensor nodes, the cluster heads are equipped with proprietary hardware extensions. In [116], PDAs are used as wireless sensor nodes in a setup for the analysis of beamforming algorithms. Further examples for the use of commodity devices in WSNs can be found in [21, 107].

However, there are a number of drawbacks, which preclude the utilization of commodity devices in wireless sensor networks. Apart from few niche applications, commodity devices do not meet WSN requirements, particularly low-cost, long lifetime, and unattended operation. The computational characteristics (like memory and CPU) of commodity devices are typically well beyond the constraints set by WSN applications. Prototypes implemented with commodity devices typically leave the question unanswered whether the same implementation could indeed be realized given more realistic cost, size, and lifetime constraints. Also, commodity devices are often too fragile for prolonged use in the physical environment and typically require extensive configuration and maintenance.

### COTS Sensor Nodes

A large class of the custom-built sensor nodes are what is referred to as COTS sensor nodes. COTS nodes are manufactured from several commercially of-the-shelf (COTS) electronic components. The aim of some COTS node developments is the provision of a versatile, broadly applicable sensor node (e.g., [21, 31, 58]), while other developments are custom-built with a particular application in mind (e.g., [61, 67]. Some of the general-purpose designs are commercially available or are made available for academic research at cost price. A typical setup consists of an RF transceiver and antenna, one or more sensors (or interfaces for connecting external sensor boards), as well as a battery and power regulating circuitry grouped around a general-purpose processor. While these processors—often 8-bit microcontrollers—typically feature some internal memory, many COTS sensor-node designs incorporate some additional external memory. The electronic components are mounted on a printed circuit board (PCB). Some COTS sensor nodes are composed of multiple PCBs, for examples, a main board and separate daughter boards for sensing and RF communication.

The computational resources provided by of most COTS-node designs are typically within reasonable limits for a number of WSN applications. Currently, the per-node cost is about 80–180 Euro (see, for example, [14, 34, 123]). Nodes built from SMD components can be as small as a one Euro coin, such as the MICA2DOT node [34]. However antennas, batteries, battery housings, the general wiring overhead of multiple components, and packaging (if required), preclude smaller sizes and form factors. The lifetime of COTS nodes ranges from days to years, depending on the dutycycle.

For many applications COTS nodes may still be too expensive to be deployed at the large scale, may be too power consuming to achieve the required network lifetime, or may be too big for unobtrusive operation. However, because of their relatively low price and availability, COTS sensor nodes allow to quickly realize prototypes, even at a larger scale. Representatives of COTS sensor nodes are the entire Berkeley Mote family (including the Rene, Mica2, MicaDot, and Mica-Z nodes [34, 57]), the iMote [31], the three generations of BTnodes [22, 73, 123], Smart-Its [124], Particle nodes [35], and many others. A survey of sensor nodes can be found in [14]. Later in this chapter we present some COTS sensor-node designs in greater detail, such as the BTnode, which has been co-developed by the author.

### Sensor-Node Systems-on-a-Chip

Research groups have recently pursued the development of entire sensor-node systems-on-a-chip (SOC). Such designs integrate most (if not all) sensor-node subsystems on a single die or multiple dies in one package. This includes microcontrollers and memories but also novel sensor designs as well as wireless receivers and transmitters. Also, chip designs have been theorized that include their own power supply, such as micro fuel cells or batteries deposited onto the actual chip. A summary of current and potential power sources and technologies for sensor nodes can be found in [99].

With their high integration levels, SOC designs usually consume less power, cost less, and are more reliable compared to multi-chip systems. With fewer

chips in the system, assembly cost as well as size are cut with the reduction in wiring overhead.

Sensor-node SOC developments are still in their early stages and have yet to be tested in field experiments. Only very few prototypes exist today. The integration level of sensor-nodes SOCs promise very-low power consumption, and very small size. The single chip design could also facilitate the easy and effective packaging. Examples of sensor-node SOCs are Smart Dust [68, 119, 120], the Spec Mote [60], and SNAP [39, 66].

### 2.3.2 Sensor-Node Components

#### Processors

Sensor node designs (both, COTS and SOC) typically feature a reprogrammable low-power 8-bit RISC microcontroller as their main processor. Operating at speeds of a few MIPS, it typically controls the sensors and actuators, monitors system resources, such as the remaining battery power as well as running a custom application. The microcontroller may also have to control a simplistic RF radio, however, more sophisticated radio transceivers include their own embedded processor for signal processing. Some applications depend on near real-time signal processing or complex cryptographic operations. Since the computational power of 8-bit microcontrollers is often too limited to perform such tasks, some sensor nodes designs use 16 or even 32-bit microcontroller, or they include additional ASICs, DSPs, or FPGAs.

However, more processing power only comes at a significant monetary price. While today 8-bit microcontrollers are about 50 US cents in very high volume (millions of parts), low-end 32-bit microcontrollers are already $ 5-7 US [103]. Pricing is particularly important for applications where the nodes are disposable and/or are deployed in high volumes. Though is can be safely expected that prices will decrease further driven by Moore's Law (to as low as $ 1 US for low-end 32-bit microcontrollers [33] in 2007), less powerful processor architectures will always be cheaper because they require to integrate less gates (i.e., transistors).

#### Memories

Sensor nodes are typically based on microcontrollers, which typically have a Harvard architecture, that is, they have separate memories for data and instructions. Most modern microcontroller designs feature integrated data and instruction memories, but do not have a memory management unit (MMU) and thus cannot enforce memory protection. Some sensor-node designs add external data memory or non-volatile memory, such as FLASH-ROM.

Microcontrollers used in COTS sensor nodes include between 8 and 512 Kbytes of non-volatile program memory (typically FLASH memory) and up to 4 Kbytes of volatile SRAM. Some additionally provide up to 4 Kbytes of non-volatile general-purpose memory, such as EEPROM. To perform memory-intensive algorithms, some designs add external data memories. The address bus of 8-bit microcontrollers is typically only 16-bit wide, allowing to address only 64 Kbytes. Therefore the entire data memory (including heap, static data

segment, and runtime stack) is constrained to this amount. The few designs with a larger memories space, organize their data memory into multiple pages of 64 Kbytes. Then one page serves as runtime memory (which holds the runtime stack and program variables) while the others can be used as unstructured data buffers. Current SOC designs typically feature RAM in the order of a few Kbytes.

Memory occupies a significant fraction of the chip real-estate. Since the die area is a dominating cost factor in chip design, memories contribute significantly to sensor-node costs. This is true for COTS microcontroller designs as well as custom-designed sensor-node SOCs. For example, 3 Kbytes of RAM of the Spec sensor-node SOC occupy 20-30% of the total die area of 1mm$^2$, not including the memory controller [60]. For comparison: the analog RF transmitter circuitry in the Spec mote requires approximately the same area as one Kbyte of SRAM, as can bee seen in Fig. 2.1 (a). In general, FLASH memory has a significant density advantage over SRAM. Modern FLASH technology produces storage densities higher than 150 Kbytes per square millimeter against the record of 60 Kbytes per square millimeter for optimized SRAM dedicated to performing specific functions [60]. Fig. 2.1 (b) shows a comparison of the occupied die sizes for 60 Kbytes of FLASH memory against 4 byte of SRAM in a commercial 8-bit microcontroller design.

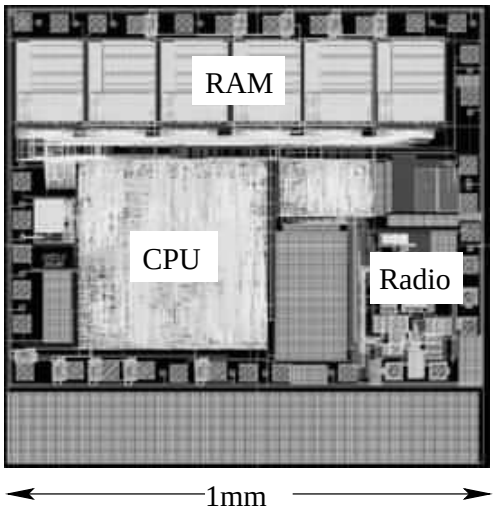
Unpaged RAM beyond 64 Kbytes also requires an address bus larger than the standard 16-bit (and potentionally memory-management units), which results in a disproportional increase in system complexity. Because of the aforementioned reasons we expect that even in the future a significant amount of all cost and size constrained sensor nodes (which we belive will be a significant amount of all sensor nodes) will not posses data memories exceeding 64 Kbytes.

### Wireless Communication Subsystems

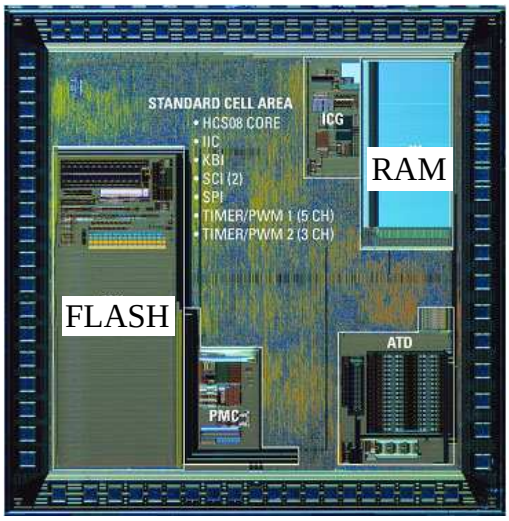
Common to all wireless sensor networks is that they communicate wirelessly. For wireless communication among the nodes of the network, most sensor networks employ radio frequency (RF) communication, although light and sound have also been utilized as physical communication medium.

***Radio Frequency Communication.*** RF communication has some desirable properties for wireless sensor networks. For example, RF communication does not require a line of sight between communication partners. The omnidirectional propagation of radio waves from the sender's antenna is frequently used to implement a local broadcast communication scheme, where nodes communicate with other nodes in their vicinity. The downside of omnidirectional propagation is, however, that the signal is diminished as its energy spreads geometrically (known as free space loss).

Various wireless transceivers have been used in sensor-node designs. Some transceivers provide an interface for adjusting the transmit power, which gives coarse-grained control over the transmission radius, if the radio propagation characteristics are known. Very simple RF transceivers, such as the TR 1000 from RF Monolithics, Inc merely provide modulation and demodulation of bits on the physical medium (the "Ether"). Bit-level timing while sending bit strings as well as timing recovery when receiving has to be performed by a host proces-




(a) Die layout of the Spec sensor-node SOC (8-bit RISC microcontroller core) with 3 Kbytes of memory (6 banks of 512 byte each) and a 900 MHz RF transmitter [60].




(b) Die layout of a commercial 8-bit microcontroller (HCS08 core from Freescale Semiconductors) with 4 Kbytes of RAM and 60 Kbytes of FLASH memory [103].

Figure 2.1: Die layouts of 8-bit microcontrollers. Memory occupies a significant fraction of the chip real-estate.

sor. Since the Ether is shared between multiple nodes and the radio has a fixed transmission frequency, it effectively provides a single broadcast channel. Such radios allow to implement the entire network stack (excluding the physical layer but including framing, flow control, error detection, medium access, etc.) according to application needs. However, the required bit-level signal processing can be quite complex and occupy a significant part of the available computing resources of the central processor. Such a radio was used in the early Berkeley Mote designs [57, 59].

More sophisticated radios, such as the CC1000 from Chipcon AS used in more recent Berkeley Mote designs, provide modulation schemes that are more resilient to transmission errors in noisy environments. Additionally, they provide a software controllable frequency selection during operation.

Finally, the most sophisticated radio transceivers used in sensor nodes, such as radio modules compliant to the Bluetooth and IEEE 802.15.4 standards, can be characterized as RF data modems. These radio modules implement the MAC layer and provide its functionality through a standardized interface to the node's main processor. In Bluetooth, for example, the physical transport layer for interface between radio and host CPU (the so-called Host Controller Interface, HCI) is defined for UART, RS-232, and USB. This allows to operate the Bluetooth module as an add-on peripheral to the node's main CPU. The BTnode, iMote, and Mica-Z sensor nodes, for example, use such RF data modems.

Some node designs provide multiple radio interfaces, typically with different characteristics. An example is the BTnode [123], which provides two short-range radios. One is a low-power, low-bandwidth radio, while the other one is a high-

power but also high-bandwidth Bluetooth radio. The ZebraNet [67] sensor node deploys a short-range and a long-range radio.

***Optical Communication.*** Light as a communication medium has radically different characteristics. One significant drawback is that because of the mostly directional propagation of light, optical communication requires a line of sight. Also, ambient light can interfere with the signal and may lead to poor link quality. Therefore optical communication may be suitable only for specific applications. However, it has several benefits over RF communication. Because of lower path loss it is more suitable for long-range communications. Also, light signals do not require sophisticated modulation and demodulation, thus optical transceivers can be realized with less chip complexity and are more energy efficient compared to radio transceivers. Also, they can be built very small.

At the University of California at Berkeley, the Smart Dust [68, 120] project examines the use of laser light for communication in their Smart Dust sensor nodes. The project aims to explore the limits of sensor-node miniaturization by packing all required subsystems (including sensors, power supply, and wireless communication) into a 1 $mm^3$ node (see Fig. 2.2 for a conceptual diagram). Optical communication has been chosen to avoid the comparatively large dimensions for antennas. A receiver for optical signals basically consists of a simple photodiode. It allows the reception of data modulated onto a laser beam emitted by another node or a base station transceiver (BST). For data transmissions, two schemes are being explored in Smart Dust: active and passive transmission. For active transmission, the sensor node generatesr a lase beam with a laser diode. The beam is then modulated using a MEMS steerable mirror. For passive transmission, the node modulates an unmodulated laser beam with a so-called corner-cube retroreflector (CCR). The CCR is a MEMS structure of perpendicular mirrors, one of which is deflectable. Incident light is reflected back to the light source, unless deflected. Several prototypes of the optical transceiver system and entire Smart Dust sensor nodes have been built. We will present one of them in the following section.

### Sensors, Sensor Boards and Sensor Interface

Application-specific sensor-node designs typically include all or most sensors. General-purpose nodes, on the other hand, include only few or no sensors at all, but provide a versatile external interface instead. This approach is beneficial since the required sensor configurations strongly depend on the target application. The external interface allows to attach various sensors or actuators directly or to attach a preconfigured sensor board. The sensors most commonly found in sensor node and sensor board designs are for visible light, infrared, audio, pressure, temperature, acceleration, position (e.g., GPS). Less typical sensor types include hygrometers, barometers, magnetometers, oxygen saturation sensors, and heart-rate sensors. Simple analog sensors are sampled by the processor via an analog-to-digital converter (ADC). More sophisticated sensors have digital interfaces, such as serial ports (e.g., USARTs) or bus systems (e.g., I2C). Over this digital interface a proprietary application protocol is run for sensor configuration, calibration and sampling. The processor of the sensor node typically implements these protocols in software. Actuators found in WSN include LEDs,

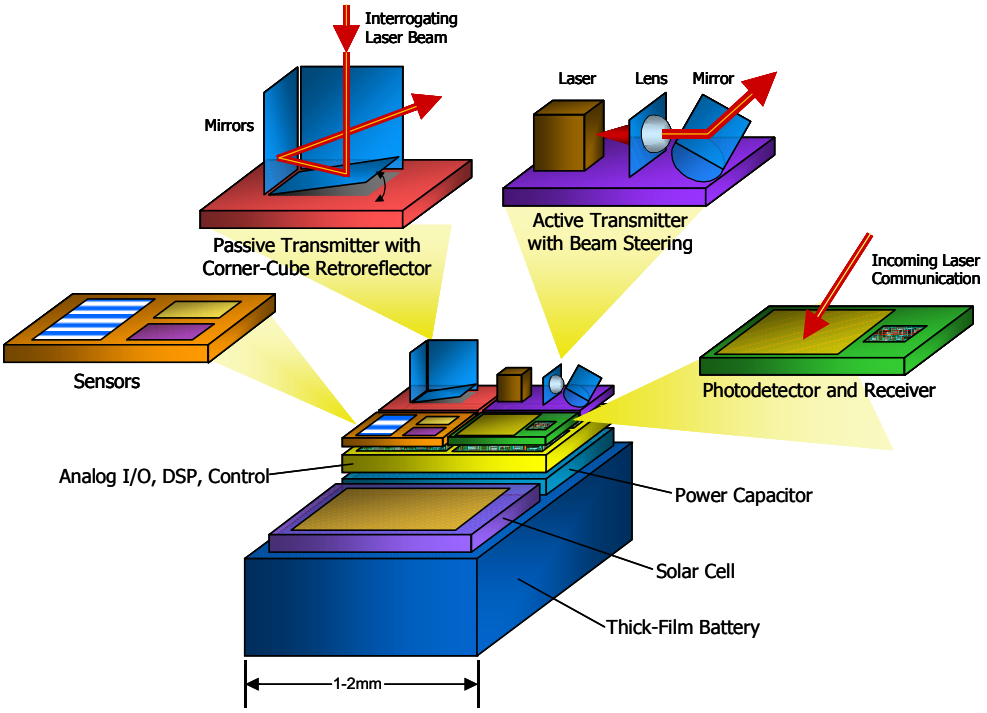



Figure 2.2: Smart Dust conceptual diagram (from [119]).

buzzers, and small speakers.

### 2.3.3 Selected Sensor-Node Hardware Platforms

Today there are a number of sensor nodes available for use in WSN research and development. Some designs have been commercialized and are available even in large numbers. Others designs have been used in limited numbers only in research projects. Still other designs are alpha versions with just a few prototypes built or which exist only in hardware simulators.

Depending on their intended application scenarios, sensor nodes have to meet vastly different (physical) application requirements, like sensor configurations and durability. However, we focus mainly on the characteristics that are relevant for developing and running software. In the following we present a few selected sensor nodes which represent various design points and different stages of maturity. An overview of available sensor node platforms can be found in [58] and in [111].

#### BTnodes

The BTnode is an autonomous wireless communication and computing platform based on a Bluetooth radio module and a microcontroller. The design has undergone two major revisions from the initial prototype. The latest generation uses a new Bluetooth subsystem and adds a second low-power radio, which is identical to those used in Berkeley Motes (see below). For a detailed description of all generations of BTnodes refer to [20].

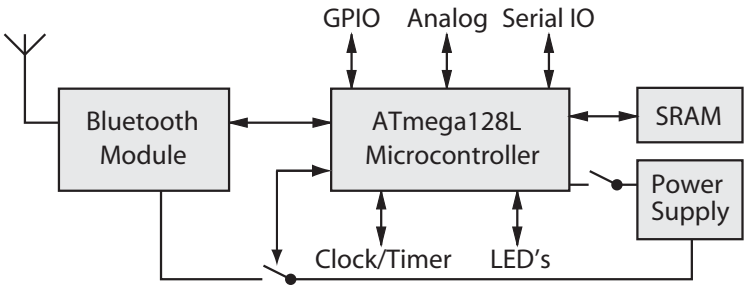



Figure 2.3: The BTnode rev2 system overview shows the sensor node's four main components: radio, microcontroller, external memory, and power supply. Peripherals, such as sensors, can be attached through the versatile general-purpose interfaces.

BTnodes have no integrated sensors, since individual sensor configurations are typically required depending on the application (see Fig. 2.3). Instead, with its many general-purpose interfaces, the BTnode can be used with various peripherals, such as sensors, but also actuators, DSPs, serial devices (like GPS receivers, RFID readers, etc.), and user interface components. An interesting property of this platform is its small form factor of 6x4 cm while still maintaining a standard wireless interface.

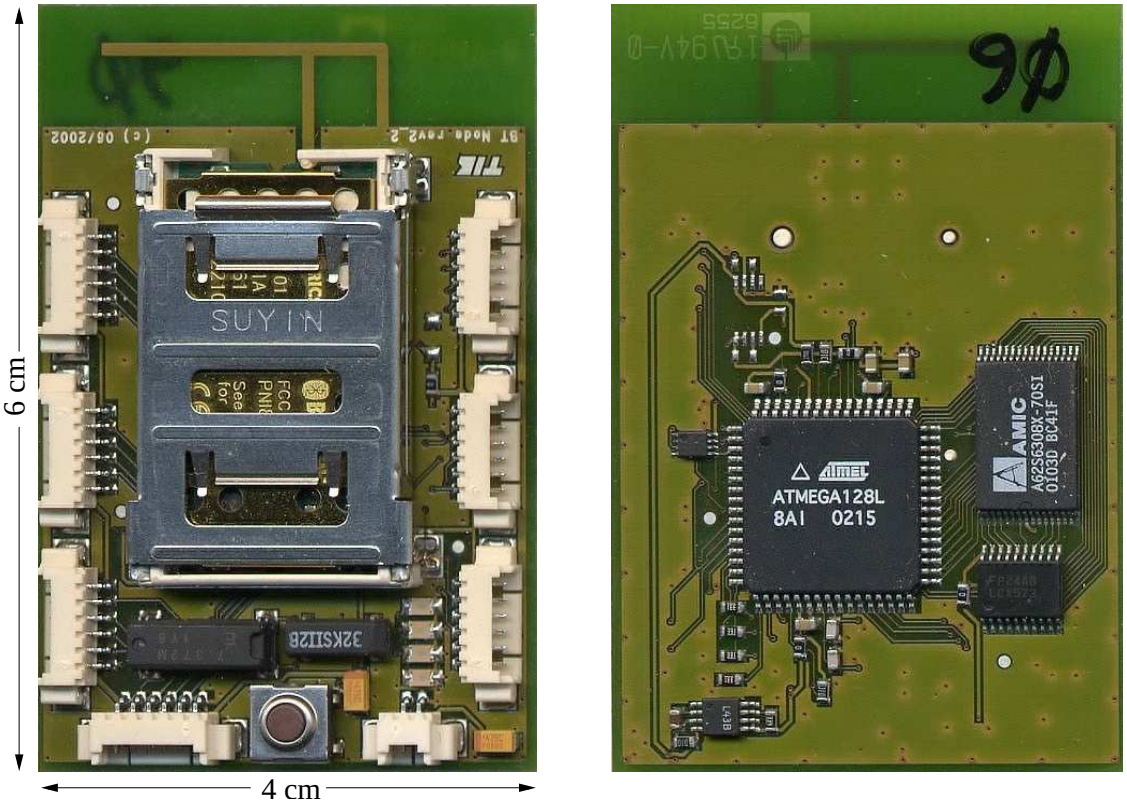



Figure 2.4: The BTnode rev2 hardware features a Bluetooth module, antenna, LEDs, and power as well as interface connectors on the front (left), and an 8-bit microcontroller and external memory on the back (right).

The second revision of the BTnode hardware (see Fig. 2.4) is built around an Atmel ATmega128L microcontroller with on-chip memory and peripherals. The

microcontroller features an 8-bit RISC core delivering up to 8 MIPS at a maximum of 8 MHz. The on-chip memory consists of 128 kbytes of in-system programmable Flash memory, 4 kbytes of SRAM, and 4 kbytes of EEPROM. There are several integrated peripherals: a JTAG interface for debugging, timers, counters, pulse-width modulation, 10-bit analog-digital converter, I2C bus, and two hardware UARTs. An external low-power SRAM adds an additional 240 kbytes of data memory to the BTnode system. A real-time clock is driven by an external quartz oscillator to support timing updates while the device is in low-power sleep mode. The system clock is generated from an external 7.3728 MHz crystal oscillator.

An Ericsson Bluetooth module is connected to one of the serial ports of the microcontroller using a detachable module carrier, and to a planar inverted F antenna (PIFA) that is integrated into the circuit board.

Four LEDs are integrated, mostly for the convenience of debugging and monitoring. One analog line is connected to the battery input and allows to monitor the battery status. Connectors that carry both power and signal lines are provided and can be used to add external peripherals, such as sensors and actuators.

### Berkeley Motes

An entire family of sensor nodes, commonly referred to as Berkeley Motes or motes for short, has been developed out of a research project at the University of California at Berkeley [61]. Berkeley Motes run the very popular TinyOS operating system. These were the first sensor nodes to be successfully marketed commercially. They are manufactured and marketed by a third-party manufacturer, Crossbow [34]. Crossbow also offers development kits and training services beyond the mere sensor devices. Because of the early availability of the hardware and because of the well maintained and documented TinyOS operating system, many applications have been realized on Berkeley Motes. They have a very active user base and are often referred to as the de facto standard platform for sensor networks.

The various members of the Berkeley Mote family (known as Mica, Mica2, Mica2Dot, Micaz, and Cricket) offer varying radio and sensor configurations at different sizes. Like the BTnode, they all feature an Atmel ATmega128L 8-bit microcontroller and allow to connect different sensor boards. Unlike the BTnode, however, they have no external SRAM but instead offer 512 Kbyte of nonvolatile memory. All Berkeley motes posses simple wireless radios with synchronous bit or packet interfaces, which require intensive real-time processing on the node's CPU.

### Smart Dust

The Smart Dust project [120] aims to explore the limits of sensor-node miniaturization by packing all required subsystems (including sensors, power supply, and wireless communication) into a 1 $mm^3$ node. The applications of such nodes are massively distributed sensor networks, such as military defense networks that could be rapidly deployed by unmanned aerial vehicles, and networks for environmental monitoring.

In 2002, a 16 $mm^3$ solar-powered prototype with bidirectional optical communication has been demonstrated in [119] (see Fig. 2.2). The system consists of three dice, which can be integrated into a single package of 16 $mm^3$. A 2.6 $mm^2$ die contains the solar-cell array, which allows the node to function at light levels of approximately one sun. The second die contains a four-quadrant corner-cube retroreflector (CCR), allowing it to be used in a one-to-many network configuration, as discussed previously. The core of the system is a 0.25 $mm^2$ CMOS ASIC containing the controller, optical receiver, ADC, photo sensor, and oscillator. This die is depicted in Fig. 2.6.

The prototype node is not programmable, but is instead operated with a 13-state finite state machine in hardware. The demonstrated node performs a fixed schedule of operations. First it toggles between sensors and initiates ADC conversion. When the conversion is complete the result is passed serially to the CCR for optical transmission. Next, the most recent byte received by the optical transceiver is echoed. Upon the completion of the transmission the cycle repeats.

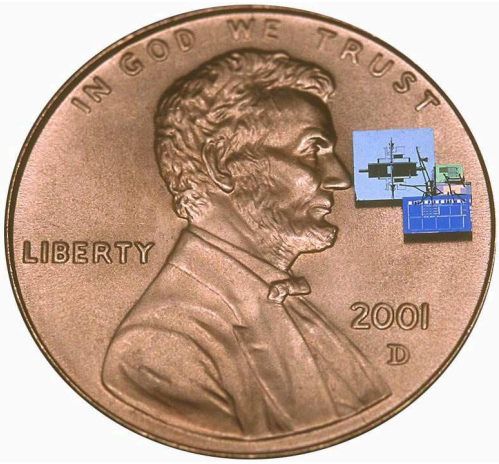
The presented Smart Dust node is a prototype with many features of an actual sensor node still missing or none-functional in a realistic environment. The solar cells do not provide enough energy to run the node in less then optimal circumstances. The light required for powering the node interferes with the optical communication. It has no packaging, so the actual node would be significantly bigger. An acceleration sensor was initially intended to be included into the design but could not due to problems with the final processing steps. Finally, the node is not programmable and thus lacks the flexibility for customization. A programmable node would occupy significantly more die space to host the programmable controller as well as data and program memories (which it currently lacks).

On the other hand, the presented node has shown to some degree what can be expected in the not-too-far future. Future motes are expected to be yet smaller through higher levels of integration, have more functionality by means of fully programmable controllers, and incorporate more types of sensors. A new development process has already been theorized (and partially tested successfully) which will yield a 6.6 $mm^3$ node with only two dice.

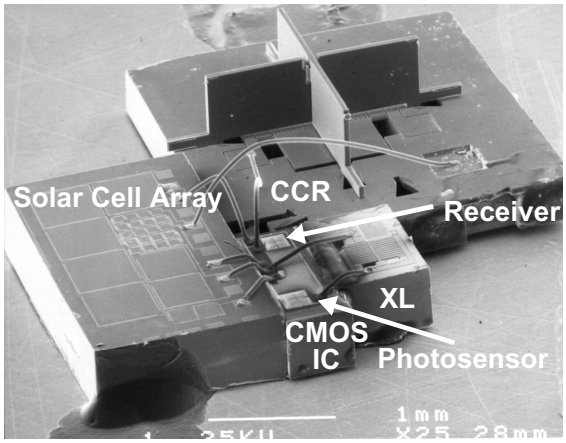
### SNAP

The sensor network asynchronous processor (SNAP) [39, 66] is a novel processor architecture designed specifically for use in low-power wireless sensor-network nodes. The design goals are low power consumption and hardware support for the predominant event-driven programming model, which is used in TinyOS and the BTnode system software.

The processor has not yet been built but low-level simulation results of the processor core exist. SNAP is based on 16-bit RISC core with an extremely low-power idle state and very low wakeup response latency. The processor instruction set is optimized to support event scheduling, pseudo-random number generation, bit-field operations, and radio/sensor interfaces. SNAP has a hardware event queue and event coprocessors, which allow the processor to avoid the overhead of event buffering in the operating system software (such as




(a) Actual size of the sensor node in comparison to a coin.




(b) Scanning electron micrograph of the node to the left.

Figure 2.5: A partially functional Smart Dust prototype-node with a circumscribed volume of 16 $mm^3$ as presented in [119].

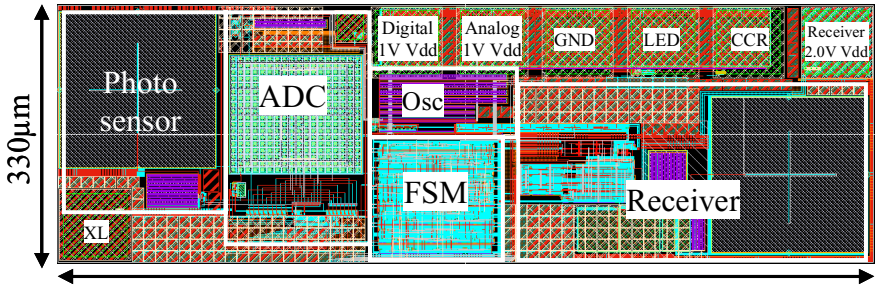



Figure 2.6: Die layout of the Smart Dust prototype presented in [119]. The detail shown is 1mm x 330um.

task schedulers and external interrupt servicing). The SNAP processor runs of a 0.6V power supply consuming about 24pJ per instruction at 28 MIPS. It features 4 Kbyte of program memory and another 4 Kbyte of data memory.

## 2.4 Embedded Systems

Wireless sensor nodes have been considered to be a subclass of embedded systems, namely *networked* embedded systems (see, for example, [4, 20, 32, 45, 57, 89]). In this section we will present general embedded-systems characteristics and relate them to those of wireless sensor nodes.

### 2.4.1 Characteristics of Embedded Systems

Unlike general-purpose computers, embedded systems are special-purpose computers engineered to perform one or a few predefined tasks. They are typically embedded into a larger device or machine, hence the name. Typical pur-

poses of embedded systems are monitoring or controlling the devices they are embedded in, performing computational operations on behalf of them, as well as providing interfaces to other systems, the environment (through sensors and actuators), and users. Often multiple embedded systems are networked, such as in automobiles and aircrafts.

### 2.4.2 Diversity of Embedded Systems

Embedded systems are very diverse, both in their application domains as well as in their requirements. Examples of embedded-system applications are industrial-machine controllers, cruise-controls and airbag controllers in automobiles, medical equipment, digital video and still cameras, controllers in washing-machines and other household appliances, fly-by-wire systems, vending machines, toys, and sensing (typically wired) monitoring systems. Depending on the application domain, embedded systems can have very diverse requirements, such as high reliability (e.g., in medical systems), high-performance data processing (e.g., image processing in cameras), low cost (e.g., in commodity devices such as remote controls), and real-time behavior (e.g., in fly-by-wire systems and airbag controllers). In fact, there does not seem to be a single set of requirements applicable to all variants of embedded systems. For example, a digital-camera controller typically requires high performance at a very low cost and low power consumption. In contrast, medical systems require high reliability and real-time behavior while performance, low cost, and low power consumption may not be an issue.

### 2.4.3 Wireless Sensor Nodes

Sensor nodes have also been considered embedded systems. Though there are some differences to traditional embedded systems, they also share many characteristics and use very similar technology. The two most obvious differences are that firstly, sensor nodes are typically not embedded into other machines or devices but are autonomous and self contained. And secondly, traditional distributed embedded systems typically use wired communication as it is more reliable.

Apart from that, sensor nodes perform tasks similar to embedded systems and have similar characteristics. Just like embedded systems, sensor nodes perform various monitoring and control tasks and often perform significant data processing. Sensor nodes also provide interfaces (to other nodes and the backend for tasking), typically through the air interface. As we have argued in Sect. 3.1, most applications of wireless sensor nodes do share a common set of requirements, probably the most important of which are resource-efficiency, reliability, and reactivity.

## 2.5 Summary and Outlook

In this chapter we have presented various applications of WSNs and their requirements on a technical hard and software solution. Often the size and cost

constrains of sensor nodes preclude the use of powerful energy supplies. To ensure adequate sensor-network lifetimes it is often necessary to strictly confine the nodes' energy consumption. As a consequence of these three restrictions (cost, size, and energy consumption), most modern sensor nodes are highly integrated and posses very little resources. Typical configurations feature 8-bit microcontrollers running at speeds of a few MIPS, tenths of kilobytes of program memory, and a few kilobytes of data memory. Though not all WSN applications necessitate such highly resource-constrained nodes, many indeed do.

The resources provisions of such integrated devices are very different compared to traditional computing systems. Also, they are used in a fundamentally different manner. WSNs are in close interaction with the environment, they are self-organized and operate without human intervention rather than relying on a fixed background infrastructure configured by system administrators. These characteristics pose new challenges on all levels of the traditional software stack.

New resource-efficient and fault-tolerant algorithms and protocols need to be developed. Classical middleware concepts need to be reconsidered in order to allow the utilization of application knowledge. New operating systems are required to economically manage the limited sensor-node resources while incurring as little resource overhead as possible themselves. Traditional operating system abstractions (such as processor sharing and dynamic and virtual memory management), which rely on hardware components of classical machine architectures (like memory management units and secondary storage) can not be easily used in sensor networks. This also requires revisiting the basic application-programming models and abstractions, such as tasks and inter-task communication. In the next chapter we will present the state-of-the-art in programming models and runtime environments for wireless sensor networks.

# 3 Programming and Runtime Environments

Developing a sensor network application typically requires to program individual sensor nodes. Today, programming of individual sensor nodes is mostly performed by programmers directly. Even though research has made much progress in distributed middleware and high-level configuration languages for sensor nodes, typically most of the application code still has to be specified by the hand of a programmer.

To support application programming, several programming frameworks exist for sensor nodes. Programming frameworks provide a standard structure to develop application programs, typically for a specific application domain, for example, Graphical User Interface programming, or, in our case, sensor-node programming. Programming frameworks for sensor nodes consist of a runtime environment with reusable code and software components, programming languages, and software tools to create and debug executable programs (see Fig. 3.1). The conceptual foundation of a programming framework is a programming model. The programming model defines the conceptual elements in which programmers think and structure their programs. These elements are operating-system abstractions (such as threads, processes, dynamic memory, etc.) and language abstractions (such as functions, variables, arguments and return parameters of procedural programming languages). Other elements are abstractions for frequently used (data) objects (such as files, network connections, tracking targets, etc.), as well as convenient operations and algorithms on those objects.

Programs specified based on high-level programming models, however, do not directly execute on the sensor-nodes hardware. Rather, they require a software layer to provide them with a runtime environment. That runtime environment bridges the gap between the high-level programming model and the low-level execution model of the processor. It typically consists of middleware components, code libraries, OS system calls, and compiler-generated code, which together provide the implementations of the model's abstractions. Naturally, a programming model with more and more expressive abstractions will require more support from the runtime environment.

The explicit goal of *traditional computing systems* is to accommodate a wide variety of applications from different domains. Therefore, they typically provide a "thick" and powerful system software that supports rich programming models and many abstractions. From this variety, application designers can (and must) choose a model that suits their applications needs best. *Sensor-node platforms*, however, are often subjected to severe resource constraints. Therefore they can afford only a thin system software layer and can thus offer only a limited programming model. The designers of programming frameworks for sensor-node

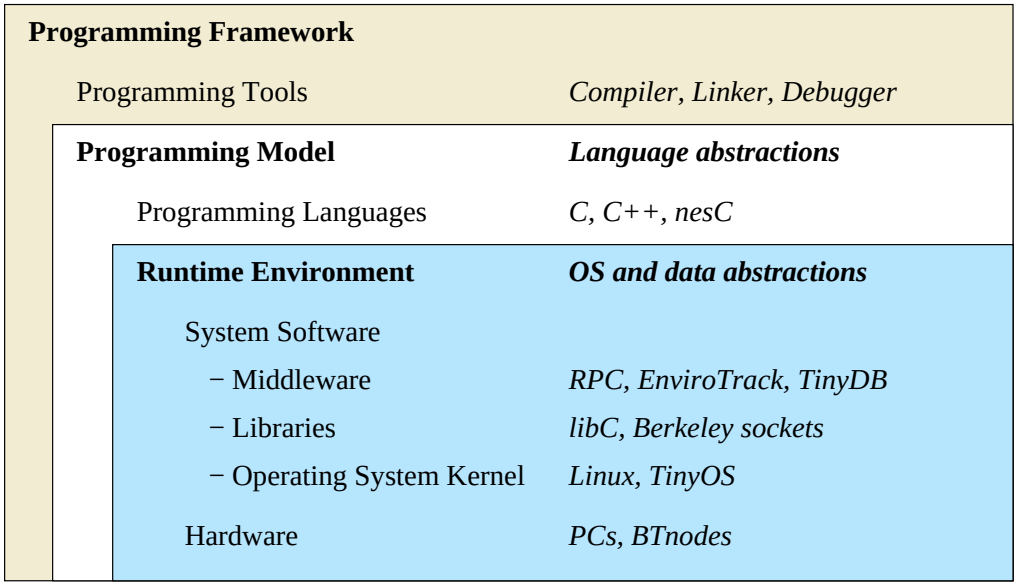



Figure 3.1: Programming frameworks for sensor nodes consist of reusable code and software components, programming languages, software tools to create and debug executable programs.

platforms must make an a priori decision which models, abstractions, and system services to support. They are faced with the dilemma to provide an expressive enough programming model without imposing too heavy a burden on the scarce system resources. Resources spent for the system software are no longer available to application programmers.

In this chapter we will review state-of-the-art programming models and the programming abstractions supported in current system software for sensor nodes. We start by presenting basic requirements that must be met by the system software and application programs alike in Sect. 3.1. Then, in Sect. 3.2, we present the three basic programming models that have been proposed for sensor-node programming, that is, the event-driven model, the multi-threaded model, and the control loop. We continue our discussion with Sections 3.3 and 3.4 on dynamic memory management and on process models, respectively. Based on our initial requirements (from Sect. 3.1) we analyze these programming models and abstractions with respect to the runtime support they necessitate. Finally, in Sect. 3.4, we summarize state-of-the-art programming frameworks (most of which we have introduced in the previous sections already in order to highlight certain aspects) and present in greater detail two programming frameworks, which represent variations of the basic event-driven model.

## 3.1 System Requirements

In the last chapter we have discussed the unique characteristics of sensor networks. These characteristics result in a unique combination of constraints and requirements that significantly influence the software running on sensor nodes. But this influence is not limited to the choice of application algorithms and data structures. The nodes' system software, that is, the operating system and other

software to support the operation of the sensor node, is affected as well. In this section we present three main requirements of WSN software. While this is not a comprehensive list of topics, it does identify the most important topics with regard to sensor-node programming.

### 3.1.1 Resource Efficiency

A crucial requirement for all WSN programs is the efficient utilization of the available resources, for example, energy, computing power, and data storage. Since all sensor nodes are untethered by definition, they typically rely on the finite energy reserves from a battery. This energy reserve is the limiting factor of the lifetime of a sensor node. Also, many sensor-node designs are severely constrained in fundamental computing resources, such as data memory and processor speeds. Sensor nodes typically lack some of the system resources typically found in traditional computing systems, such as secondary storage or arithmetic co-processors. This is particularly the case for nodes used in applications that require mass deployments of unobtrusive sensor nodes, as these applications prompt small and low-cost sensor-node designs.

In addition to energy-aware hardware design, the node's system software must use the available memory efficiently and must avoid CPU-cycle intense operations, such as copying large amounts of memory or using floating-point arithmetic. Various software-design principles to save energy have been proposed [41]. On the algorithmic level, localized algorithms are distributed algorithms that achieve a global goal by communicating with nodes in a close neighborhood only. Such algorithms can conserve energy by reducing the number and range of radio messages sent. Adaptive fidelity algorithms allow to trade the quality of the sensing result against resource usage. For example, energy can be saved by reducing the sampling rate of a sensor and therefore increasing the periods where it can be switched off.

Resource constraints clearly limit the tasks that can be performed on a node. For example, the computing and memory resources of sensor nodes are often too limited to perform typical signal processing tasks like FFT and signal correlation. Hence, clustered architectures were suggested (e.g., [117]), where the clusterheads are equipped with more computing and memory resources leading to heterogeneous sensor networks.

Resource constraints also greatly affect operating system and language features for sensor nodes. The lack of secondary storage systems precludes virtual-memory architectures. Also, instead of deploying strictly layered software architectures, designers of system software often choose cross-layer designs to optimize resource efficiency. Programming frameworks for sensor nodes should support application designers to specify resource efficient and power-aware programs.

### 3.1.2 Reliability

Wireless sensor networks are typically deployed in remote locations. As software failures are expensive or impossible to fix, high reliability is critical. Therefore, algorithms have been proposed that are tolerant to single node and net-

work failures. However, immanent bugs in the nodes' system software may lead to system crashes that may render an entire WSN useless. Particularly if many or all nodes of the network run identical software or if cluster heads are affected, the network may not recover.

Programming frameworks for sensor nodes should support application designers in the specification of reliable and, if possible, verifiable programs. Some of the standard programming mechanisms, like dynamic memory management and multi-threading, are known to be hard to handle and error prone. Therefore some programming frameworks deliberately choose not to offer these services to the application programmer.

### 3.1.3 Reactivity

The main application of WSNs is monitoring the real world. Therefore sensor nodes need to be able to detect events in the physical environment and to react to them appropriately. Also, due to the tight collaboration of the multiple nodes of a WSN, nodes need to react to network messages from other nodes. Finally, sensor nodes need to be able to react to a variety of internal events, such as timeouts from the real-time clock, interrupts generated by sensors, or low battery indications.

In many respects sensor nodes are *reactive systems*. Reactive systems are computer systems that are mainly driven by events. Their progress as a computer system depends on external and internal events, which may occur unpredictably and unexpectedly at any time, in almost any order, and at any rate. Indeed, the philosophy of sensor-node programming frameworks mandates that computations are only performed in reaction to events and that the node remains in sleep mode otherwise in order to conserve power.

On the occurrence of events, wireless sensor nodes react by performing computations or by sending back stimuli, for example, through actuators or by sending radio messages. Sensor nodes need to be able to react to all events, no matter when and in which order they occur. Typically reactions to events must be prompt, that is, they must not be delayed arbitrarily. The reactions to events may also have real-time requirements. Since the handling of events is one of the main tasks of a sensor node, programming frameworks for sensor nodes should provide expressive abstractions for the specification of events and their reactions.

## 3.2 Programming Models

Sensor nodes are reactive systems. Programs follow a seemingly simple pattern: they remain in sleep mode until some event occurs, on which they weak up to react by performing a short computation, only to return to sleep mode again. Three programming models have been proposed to support this pattern while meeting the requirements specified previously. These models are the event-driven model, the multi-threaded model, and the control-loop model. The event-driven model is based on the event-action paradigm, while the multi-threaded model and the control loop are based on polling for events (in a block-

ing and non-blocking fashion, respectively).

### 3.2.1 Overview

In the *control-loop model* there is only a single flow of control, namely the control loop. In this loop, application programmers must repeatedly poll for the events that they are interested in and handle them on their detection. Polling has to be non-bocking in order not to bring the entire program to a halt while waiting for a specific event to occur. Then, the computational reaction to an event must be short in order not to excessively delay the handling of next events.

In the *multi-threaded model*, programmers also poll for events and handle them on detection. However, polling can be blocking since programs have several threads with individual flows of control. For the same reason the duration of a computational reaction is not bounded. When one thread is blocking or performing long computations, the operating system takes care of scheduling other threads concurrently, that is, by sharing the processor in time multiplex.

In the *event-driven model* the system software remains the program's control flow and only passes it shortly to applications to handle events. The system software is responsible to detect the occurrence of an event and then to "call back" a specific event-handling function in the application code. These functions are called actions (or event handlers). They are the basic elements of which applications are composed of. Only within these actions do applications have a sequential flow. The scheduling of actions, however, is dictated by the occurrence of events.

These models differ significantly in their expressiveness, that is, the support they provide to a programmer. They also require different levels of support from the runtime environment and thus the amount of system resources that are allocated for the system software. In the following we will analyze these three models regarding their expressiveness and inherent resource consumption.

### 3.2.2 The Control Loop

Some sensor-node platforms (such as the Embedded Wireless Module nodes described in [97] and early versions of Smart-Its nodes [124]) provide programming support only as a thin software layer of drivers between the hardware and the application code. Particularly, these systems typically do not provide a programming abstraction for event-handling or concurrency, such as actions or threads. Rather they only provide a single flow of control.

In order to guarantee that a single-threaded program can handle and remains reactive to all events, the program must not block to wait for any single event. Instead, programmers typically poll for the occurrence of events in a non-blocking fashion. Among experienced programmers it is an established programming pattern to group all polling operations in a single loop, the so-called *control loop*. (In this respect the control-loop model is much more a programming pattern or approach for sensor nodes–or, more generally, for reactive systems–rather than a programming model with explicitly supported abstractions.) The control loop is an endless loop, which continuously polls for the occurrence of events in a non-blocking fashion. If the occurrence of an event

has been detected, the control loop invokes a function handling the event. For optimizations actual programs often depart from this pattern. The pseudo-code in Prog. 3.1 depicts the principle of a control loop.

Program 3.1: A simple control loop.

```
1 while( true ) { // control loop, runs forever
2   if( status_event_1 ) then event_handler_1();
3   if( status_event_2 ) then event_handler_2();
4   //...
5   if( status_event_n ) then event_handler_n();
6 }
```

Programming frameworks supporting a control loop as their main approach to reactive programming have several benefits. Firstly, they require no concurrency and event-handling support from an underlying runtime system, such as context switches or event queuing and dispatching. The only support required are drivers that allow to check the status of a device in a non-blocking fashion. Therefore, their runtime environments are very lean. Secondly, programmers have full control over how and when events are handled. The entire control flow of the program is exclusively managed by application programmers and can thus be highly optimized.

On the other hand, the control-loop approach does not *enforce* a particular programming style. Programmers carry the burden to implement event detection (i.e., polling operations), queuing and dispatching without being guided along a well approved and established path. This burden can be hard to master even for expert programmers. Particularly when event detection and event handling are not clearly separated in a control loop, the model's high flexibility can lead to code that is very unstructured and therefore hard to maintain and extend. Such code typically depends on timing assumptions and is non-portable. Also, programmers are responsible for implementing sleep modes to avoid busy-waiting when no events occur and need to be processed. Implementing sleep modes is particularly troublesome when event detection and handling are not clearly separated.

### 3.2.3 Event-driven Programming

The event-driven programming model can be seen as a conceptualization of the control-loop approach. It does enforce the separation of event detection and event dispatching by making the control loop an integral part of its runtime environment. The event-driven model is the most popular programming model for developing sensor-network applications today. Several programming frameworks based on this model exist, for example, our BTnode system software [21], but also the popular TinyOS and nesC [45], Contiki [37], SOS [52] and Maté [79].

### The Basic Programming Model

The event-driven model is based on two main abstractions: events and actions. Events represent critical system conditions, such as expired timers, the reception of a radio message, or sensor readouts. Events are typically typed to distinguish different event classes and may contain parameters, for example, to carry associated event data. At runtime, the occurrence of an event can trigger the execution of a computational action. Typically, there is a one-to-one association between event types and actions. Then, an event-driven program consists of as many actions as there are event types (see pseudo code in Prog. 3.2).

Actions are typically implemented as functions of a sequential programming language. Actions always run to completion without being interrupted by other actions. A so-called *dispatcher* manages the invocation of actions. Dispatchers often use an event queue to hold unprocessed events. The internal implementation of dispatchers typically follows the control-loop approach.

Event-driven systems are typically implemented in a single flow of control. (Indeed, in sensor networks they are often deployed just to avoid the overhead of concurrent systems.) In order not to monopolize the CPU for any significant time and thus allow other parts of the system to progress, actions need to be non-blocking. Actions are expected to terminate in bounded time—typically less than a few milliseconds, depending on the application's real-time requirements and the system's event-queue size. Therefore, at any point in the control flow where an operation needs to wait for some event to occur (e.g., a reply message or acknowledgment in a network protocol), the operation must be split into two parts: a non-blocking operation request and an asynchronous completion event. The completion event then triggers another action, which continues the operation.

Event-driven programs consist of actions, each of which is associated to an event and which is invoked in response to the occurrence of that event. To specify a program, programmers implement actions (typically as functions of a sequential language). At system start time, control is passed to the dispatcher. The control then remains with the system's dispatcher and is only passed to application-defined actions upon the occurrence of events. After the execution of an action, control returns to the dispatcher again. Therefore, from the programmer's point of view, applications consist of a number of distinct program fragments (i.e., the actions). These fragments are *not* executed in a given sequential order, as functions of procedural languages would. Rather, these actions are invoked in the order in which their associated events occur.

### Discussion

The main strength of the event-based model is the enforcement of a well defined and intuitive programming style which incurs only little overhead. The event model is simple, yet nicely captures the reactive nature of sensor-network applications. It enforces the by sensor-network experts commonly promoted programming pattern, in which short computations should only be triggered by events. Compared to the control-loop approach, programs in the event-driven model are clearly structured into a few well-defined actions. Furthermore, programmers must not concern themselves with sleep modes, as they are typically

Program 3.2: The general structure of an event-driven program.

```
1 void main() {
2   // initialize dispatcher, runs forever
3   start_dispatcher();
4   // not reached
5 }
6 void event_handler_1( EventData data ) { // associated with event_1
7   // initialize local variables
8   process( data );                      // process event data
9 }
10 void event_handler_2() { ... }         // associated with event_2
11 ...
```

provided as basic service of the system software's dispatcher.

Supporting events in the underlying system software incurs only very little overhead. The system software must only provide basic services for event detection, queuing, and dispatching. It has been shown in several prototypes that these elements can be implemented very resource efficiently in software (see, for example, [37, 52, 59, 73]). To further reduce resource and energy consumption, much of the model's runtime system can be implemented in hardware, as has been shown in the SNAP processor architecture [39, 66]. Event-driven application programs can thus run in very lightweight execution environments.

As a consequence, in the context of sensor networks, event-based systems are very popular with several programming frameworks in existence. Today, these frameworks are supported by a large user base, most notably TinyOS. Hence, many programmers are used to think in terms of events and actions and find it natural to structure their programs according to this paradigm.

### 3.2.4 Multi-Threaded Programming

The multi-tasking model embodies another view on application programming. The multi-threading programming model focuses on describing application programs as a set of concurrent program fragments (called threads), each of which has its own context and control flow. Each thread executes a sequential program, which may be interrupted by other threads. Since wireless sensor nodes are typically single-processor systems, concurrency is only a descriptive facility. On the node's processor, concurrent threads are scheduled sequentially. All threads of a program share resources, such as memory. Shared memory is typically used for inter-thread communication.

#### The Basic Programming Model

The multi-threaded programming model is convenient for programming reactive sensor-node applications. Threads may block to await the occurrence of certain events, for example, the reception of a radio message. A thread waiting for a particular event becomes active again only after the occurrence of the

event. Therefore, programmers can simply wait for the occurrence of particular events in a *wait statement* and handle the reaction to that event (in the same computational context) when the statement returns (see Prog. 3.3). To handle the different events that are of interest to an application, programmers typically use several threads, one per event or group of related events. While a thread is blocking, the operating system schedules other threads, which are not waiting.

Programmers need not worry about the reactivity of the program when a thread blocks or when it performs long-running operations, as the rest of the application remains reactive and continues to run. For this reason, graphical user interfaces (GUIs) of PC applications are often implemented in their own thread. In wireless sensor networks, threads are typically used to specify the subsystems of a sensor node, such as network management, environmental monitoring, and monitoring the node's local resources. Programming loosely coupled parts of an application as separate threads can increase the structure and modularity of the program code.

In preemptive multi-threading systems, a thread of execution can be interrupted and the control transferred to another thread at any time. Multiple threads usually share data, thus requiring synchronization to manage their interaction. Synchronization between threads must ensure deterministic access to shared data, regardless how threads are actually scheduled, that is, how their threads of execution are interleaved. Proper synchronization prevents data inconsistencies when concurrent threads simultaneously modify and access shared data. Synchronization is often implemented using semaphores or by enforcing atomic execution (by disabling context switches).

Program 3.3: The structure of multi-threaded sensor-node program.

```
1  void main() {
2    // initialization of two threads
3    start( thread_1 );
4    start( thread_2 );
5  }
6  void thread_1() {
7    // initialization of local variables
8    event_data_t ev;
9    while( true ) {          // do forever
10     waitfor( EV_1, ev );   // wait for event of type EV_1
11     process( ev.data );    // process event data
12     waitfor( EV_2, ev );   // wait for event of type EV_2
13     process( ev.data );    // process event data
14     // ...
15   }
16 }
17 void thread_2() { ... }
```

### System Requirements for Multi-Threading

Multi-threading requires operating-system level support for switching between the contexts of different tasks at runtime. In a *context switch*, the currently running thread is suspended by the system software and another thread is selected for execution. Then the context of the suspended thread must be saved to memory, so that it can be restored later. The context contains the program state as well as the processor state, particularly any registers that the thread may be using (e.g., the program counter). The data-structure for holding the context of a thread is called a *switchframe*. The system software must then load the switchframe of the thread to run next and resume its execution.

Multi-threading requires significant amounts of memory to store the switchframes of suspended threads. Also, in every context switch, data worth of two switchframes is copied, from the registers to data memory and vice versa. For slow and resource-constrained sensor nodes this means a significant investment in memory as well as CPU cycles [36, 59].

For example, the Atmel ATmega 128 microcontroller, a microcontroller used in several COTS sensor-node designs (e.g., Berkeley Motes, BTnodes, and others), has 25 registers of 16 bit each. Therefore each context switch requires to copy 100 bytes of data only for saving and restoring the processor state. Experiments on this microcontroller in [36] confirm that the context switching overhead of multi-threaded programs is significant, particularly under high system load. A test application run under high load (i.e., under a duty cycle of about only 30%) on the multi-threaded Mantis operating system exhibited 6.9% less idle time compared to a functionally equivalent application on the event-driven TinyOS. (Before both operating systems had been ported to the same hardware platform). The reduction in idle time and the associated increase in power consumption are attributed to the switching overhead.

Additionally, each thread has its own state, which is stored by the runtime stack. The runtime stack stores information about the hierarchy of functions that are currently being executed. Particularly it contains the parameters of functions, their local variables, and their return addresses. The complete per-thread stack memory must be allocated when the thread is created. It cannot be shared between concurrent threads. Because it is hard to analyze how much stack space a thread needs exactly, runtime per-thread stacks are generally over-provisioned. The consequence of under-dimensioned stacks are system crashes at runtime.

The overhead of per-thread data-structures, that is, runtime stacks and switchframes, is often forgotten in evaluations of memory footprints of multi-threaded sensor-node operating systems. However, this inevitable memory overhead can consume large parts of the memory resources of constrained nodes (cf. [37]).

### Reliability, Debugging, and Modularity Issues

Besides of its memory overhead, multi-threading has issues with reliability and modularity. Programmers are responsible for synchronizing threads manually using special synchronization primitives, which materializes as additional program code. Proper synchronization requires a comprehensive understanding of the data and timing dependencies of all interacting threads within a pro-

gram. Gaining this understanding becomes increasingly difficult in programs with many and complex dependencies, particularly when multiple programmers are involved.

It is generally acknowledged that thread-synchronization is difficult, because threads may interact with each other in unpredictable ways [30, 94]. Common errors are data races and deadlocks. Data races occur because of missing thread synchronization. However, also the overuse of synchronization primitives can lead to errors, namely deadlocks, and execution delays. Particularly atomic sections in application programs can delay time critical operations within drivers and lead to data loss. Such errors may break the application program as well as even the most carefully crafted error-recovery mechanisms, rendering affected sensor nodes useless in the field. Choi et al. [30] have found that programmers that are new to multi-threading often overuse locking primitives.

On top of being error-prone, multi-threaded programs are also very hard to debug. Often synchronization errors depend on the concrete timing characteristics of a program [30]. Local modifications to the program code can expose a previously undetected error in a different part of the code. In the case of an error typically the exact timing history cannot be reproduced to aid in debugging. These characteristics make tracking down and fixing errors painful and time consuming.

Finally, multi-threading may also hinder program modularity. Threads typically communicate with shared data. Because of the synchronization required for the protection of shared data, interacting threads are no longer independent of each other. To reuse threads in a different program context or on a different hardware platform programmers must understand their internals, such as data dependencies and timings, and carefully integrate them into the new program context. Therefore threads cannot be designed as independent modules and have no well defined interfaces.

### Discussion

Though threads are a powerful programming abstraction (and particularly more powerful than the event/action abstraction of the event-driven model), its power is rarely needed. Because of its reliability and modularity issues, it has been suggested [94] that multi-threaded programming in general (i.e., even for general purpose computing platforms) should only be used in the rare cases when event-driven programming does not suffice. Particularly for programs that have complex data dependencies, thread synchronization becomes too difficult to master for most programmers. On the contrary, due to the run-to-completion semantics, synchronization is rarely an issue in event-driven programming. Additionally, the event-driven model incurs less overhead.

As we will discuss in the next chapter, WSN applications typically have many and complex data dependencies throughout all parts of the program. Because of the severe reliability and modularity issues that are thus to be expected and the high resource overhead, we believe that the multi-threaded programming model is not suitable as the main programming model for particularly resource-constrained sensor nodes. This belief concurs with the view of many sensor-network system experts (cf. [59, 60] and others). Most notably the designers of

the Contiki programming framework [37], which combines the multi-threaded and event-driven programming model, suggest that threads should be used only as a programmer's last resort.

## 3.3 Memory Management

In general-purpose programming frameworks there are two basic mechanisms to memory management, which are typically used both. *Dynamic memory management* provides programmers with a mechanism to allocate and release chunks of memory at any point in the program during runtime. In the absence of dynamic memory management (i.e., with *static* memory management), chunks of memory can only be allocated at compile time. These chunks can neither be released nor can their size be changed at runtime. Dynamic memory management does not depend on a particular programming or process model (as discussed in the next section)—it can be used in combination with any model.

The second mechanism for the allocation of (typed) data is *automatic variables* (typically also called local variables). Automatic variables are tied to the scope of a function in sequential languages and are typically allocated on the stack. When the function ends, automatic variables are automatically released (i.e., without the manual intervention of a programmer). Automatic variables are considered an integral part of sequential (i.e., procedural) programming languages. They are provided by all sensor-node programming frameworks based on procedural languages known to us. On the other hand, dynamic memory management has several implementation and reliability issues, which has lead to its exclusion in many sensor-node software designs. In this section we will briefly discuss these issues.

### 3.3.1 Resource Issues

Today, dynamic memory management (as provided, for example, by the traditional C-language API `malloc()` and `free()`), is taken for granted by most programmers. However, dynamic memory management is a major operating-system service and requires thoughtful implementation. The memory allocation algorithm must minimize the fragmentation of allocated memory chunks over time but must also minimize the computational effort for maintaining the free-memory list. Implementations of memory-allocation algorithms can constitute a significant amount of the code of a sensor node's system software. Also, the data structures for managing dynamic memory can consume significant amounts of memory by themselves, particularly if arbitrary allocation sizes are supported.

### 3.3.2 Reliability Concerns

In addition to the high cost of implementing dynamic memory management, there are several reliability concerns. Typically coupled with virtual memory and memory protection, dynamic memory management in traditional systems relies on the support of a dedicated piece of hardware, the Memory Management Unit (MMU). Since resource-constrained sensor nodes do not have a MMU, they

do not support memory protection. Thus the private data of user applications and the operating system cannot be protected mutually from erroneous write access.

Besides a general susceptibility to memory leaks, null-pointer exceptions, and dangling pointers, there is an increased concern to run out of memory. Due to the lack of cheap, stable, fast, and power efficient secondary storage, non of the sensor-node operating systems known to us provide virtual memory. Instead, sensor-node programs have to rely on the limited physical memory available on the node.

An application's memory requirements need to be carefully crafted to under no circumstances exceed the available memory (or to provide out-of-memory error handlers with every memory allocation). However, crafting an application's memory requirements is considered impractical or at least very hard for systems supporting concurrency and for large applications that grow over time and which involve multiple developers. Concurrent programs and threads compete for the available memory. It is hard to reason about the maximum memory requirements of a program as that would require to test every single control path through the application for memory allocations.

While memory leaks, null-pointer exceptions, and dangling pointers could be possibly debugged with the help of specialized development tools, it should be noted that there are no practical techniques to reason about the entire memory requirements (including stack size and dynamic memory) of an application. In order to avoid out-of-memory errors, either the physical memory installed on a node must be greatly over-provisioned (in terms of the program's expected maximum memory usage) or the developer relies entirely on static memory allocation (plus other safety measures to prevent unlimited stack growth, like prohibiting recursion).

### 3.3.3 Dynamic Memory Management for Sensor Nodes

Many sensor-node programmers have been hesitant to use dynamic memory due to the mentioned implementation and reliability issues [48]. Therefore, many operating systems for sensor nodes do not provide support for dynamic memory management, for example, TinyOS [45], the BTnode system software [21], and BTnut [123]. Currently MANTIS [8] does not provide dynamic memory management but it is scheduled for inclusion in future versions.

Other operating systems for sensor nodes, like SOS [52], Impala [81], and SNACK [48], provide dynamic memory management, but, in order to avoid fragmentation of the heap, use fixed block sizes. Fixed-size memory allocation typically results in the allocation of blocks that are larger than needed and therefore wastes a precious resource. In SNACK [48] only buffers for network messages are dynamically allocated from a managed buffer pool. Variables have to by allocated statically.

## 3.4 Process Models

Besides a basic programming model, the system software also provides other features to support normal operation, such as process management. Processes can be thought of as programs in execution. Two important features found in most traditional system software related to process management are the ability to run several processes concurrently and the ability to create processes dynamically. It has been argued that these features would also be useful for sensor networks, particularly in combination. For example, software maintenance and changing application needs may require updates to a node's software at runtime [52], that is, after the node has been deployed and has started program execution. It may also be useful to start and run applications in parallel [25], for example, to run short-term tests or to perform sensor recalibration without disrupting long-running monitoring applications.

Dynamically loadable programs and process concurrency are independent of each other and require different support mechanisms in the system software. Here again, for reasons of resource-efficiency and reliability, system designers have to make design tradeoffs. In this section we will first present the runtime support required for each feature before analyzing them together.

### 3.4.1 Overview

Operating systems supporting a *dynamic process model* can load a program's executable (e.g., from the network or from secondary storage) at runtime to create a new process. They can also terminate a process at runtime and remove its executable from the system's memories. In contrast to this, in a *static process model*, the system has to be shut down and reconfigured before a new application can be run or removed from the system. Since physical contact with the node is often infeasible after deployment of the WSN, several sensor-node platforms provide a mechanism for delivering executables over-the-air through the wireless interface. A dynamic process model is most useful and therefore often combined with a *concurrent process model*, where several programs can run "at the same time" by sharing the processor in time-multiplex.

### 3.4.2 Combinations of Processes Models

Modern general-purpose operating systems combine dynamic and concurrent process models. In contrast, several sensor-node platforms, like TinyOS, BTnut, and the BTnode system software, only provide a static, non-concurrent process model. All existing sensor-node platforms that do provide dynamic processes loading, however, also provide an environment for their concurrent execution.

#### Dynamic and Concurrent Process Models

In a dynamic process model, individual processes can be created at runtime. To support dynamic loading of a process at runtime the program's binary image must be loaded (e.g., from secondary storage or over the network) into the

node's different memories. Process creation involves loading the program's executable code (known as the text section) into the node's program memory, while statically allocated and initialized data (known as the data section, which contains, for example, global and static variables) are copied to data memory. At compile time the exact memory locations where the program will reside after loading is unknown. Therefore, the compiler generates so-called relocatable code, where variables and functions are addressed by relative memory locations only. The system software then replaces the relative memory locations with absolute addresses when the program is loaded at runtime. This process is called *relocation*. In a combined, dynamic and concurrent program model, the newly created process is added to the list of currently running processes. The system software often is a separate process). For example, system services in Contiki [37] are implemented as processes that can be replaced at runtime.

#### Static and Non-Concurrent Process Model

On the other hand, in a *static process model* in combination with a non-concurrent process model only a single program can be installed and running at any time. In such systems, the application code is linked to the system software at compile time, resulting in a monolithic executable image. The executable image is then uploaded to the memories of the node, effectively overwriting any previous software (application code as well as system-software code). A relocation step is not required.

To be able to change the node's software despite a static process model, several runtime environments allow to reload the entire monolithic system image. To do so, the currently running software stops its normal operation to receive the new software image from the network and then saves it to data memory. Then it reboots the processor in a special bootloader mode. The bootloader finally copies the image from data memory to program memory and reboots the processor again in normal mode to start the newly installed image.

#### Other Combinations

Other combinations (concurrent but static; dynamic but non-concurrent) are also possible, but have not seen much use in sensor-node operating systems. Only the Maté virtual machine [79] allows to run a single, non-concurrent process, which can be dynamically replaced during runtime.

### 3.4.3 Over-the-Air Reprogramming

To allow software updates of nodes deployed in the field, several systems implement *over-the-air reprogramming*, where a new program image can be downloaded over the wireless interface. Over-the-air reprogramming should not be confused with a dynamic process model. Over-the-air reprogramming merely denotes the systems capability to receive a program image over the air and to initiate its loading. The actual loading of the executable image to the system's memories then still requires one of the loading procedure as described above.

Actual implementations of over-the-air programming often require the cooperation of user-written application code. Over-the-air reprogramming is often provided as a substitute for dynamic process loading by several resource-constrained sensor-node architectures. It has been deployed, for example, in BTnode system software [21], BTnut [123], TinyOS [45]), Maté [79], and others. A detailed survey of software update management techniques can be found in [51].

### 3.4.4 Process Concurrency

In a concurrent process model, several processes can run at the same time. Since wireless sensor nodes are typically single-processor systems, concurrent processes need to be interleaved, with the processor multiplexed among them. Processor multiplexing is called *scheduling* and is typically performed by the system software as it is considered a fundamental system service. The points in the programs where switching is best performed depends on the programs' internal structure. Therefore, scheduling mechanisms are closely related to programming models and their corresponding execution environments. For sensor nodes two basic scheduling strategies exist. Architectures with an event-driven execution environment typically provide a concurrency model that is also based on events. Systems with a multi-threaded execution environment provide a concurrency model based on context switches.

#### Context-Switching Processes

The design of process concurrency based on context switches for sensor nodes can be best understood by comparing it to designs in general-purpose operating systems.

***Multi-tasking in general-purpose operating systems.*** Modern general-purpose operating systems have two levels of concurrency—the first level among processes, and the second level among the threads within a process. Both levels have their own scheduling mechanism, which are both based on context switches. Context-switching processes is referred to as *multi-tasking*. (The terms task and process are often used synonymously.)

Multi-threading and multi-tasking are similar in many respects but differ in the way they share resources. They both provide concurrency, the former among the threads of a process, the latter among the processes running on a system. Just like threads, tasks have their own sequential control flow and their own context. And just like multi-threading, multi-tasking requires context switches and individual stacks (one per task) to store context information. Therefore, the system support required for multi-tasking is also very similar to that of multi-threading, which we discussed previously in Sect. 3.2.4 on page 38.

The main difference to threads is that tasks typically do not share resources, such as files handles and network connections. In particular, tasks have their own memory regions that are protected against accidental access from other tasks. This is desirable to protect correct programs from buggy ones. To provide communication and synchronization amongst processes despite those protec-

tive measures, special mechanisms for inter-process communication exist, like shared memory, message passing, and semaphores.

The main objectives of process concurrency in general-purpose operating systems is to optimize processor utilization and usability (cf. [108]). While one process is waiting for an input or output operation to commence (e.g., writing a file to secondary storage or waiting for user input), another process can continue its execution. Therefore some process is always running. From a user perspective, process concurrency can increase the systems usefulness and reactivity. Running several programs in parallel allows to share the processing power of a single computer among multiple users. To each of the users it appears as if they had their own (albeit slower) computer exclusively to them. Also, a single user running several applications can switch back and forth between them without having to terminate them.

***Context-switching sensor-node processes.*** In order to avoid the resource overhead induced by two layers of scheduling, context-switsched sensor-node operating systems typically only provide a single level of scheduling on the basis of threads. That is, threads are the only scheduled entity. This does not mean, however, that multi-threaded sensor-node operating systems cannot provide process concurrency. If they do, the threads of distinct processes are context-switched, thereby also switching processes. Processes then consist of a collections of threads, which are indistinguishable from the threads of other processes. The notion of a process is entirely logical, denoting all threads belonging to a program. In simple implementations a fine-grained control mechanism over the CPU time assigned to processes is missing; a process with more threads may receive more CPU time. To dynamically create a new process, the binary image of the program (consisting of a collection of threads) is loaded and all of its threads are instantiated.

***Threads or tasks?*** In the face of a single level of context switching, the question my arise, why we consider the context-switched entities of sensor-node system software to be threads rather then processes. This seems to be a rather arbitrary, if not an unusual view, because in the history of modern operating-system development, support for concurrent processes (e.g., multi-tasking) was introduced well before concurrency within individual processes, as provided by multi-threading (cf. [27]). In fact, multi-threading was long missing in general-purpose operating systems.

Though threads and tasks are similar in several respects, they have been developed for different reasons and serve different objectives. While multi-tasking aims more at the user of the computer system, multi-threading is mainly an abstraction to aid programmers. In sensor networks, concurrency is mainly considered a tool for specifying reactive programs while supporting multiple processes or even multiple users is considered less important. Also, the context-switched entities in wireless sensor networks have much more in common with threads than processes, because of the way they share resources. Most of the measures to protect distinct processes found in traditional multi-tasking systems are not (and cannot be) implemented in sensor networks. Because of these reasons, the term task has become widely accepted in sensor-network literature to denote context-switched entities.

### Event-based Process Concurrency

In programming frameworks based on the event-driven programming model, programs are composed of several actions. A property of the event-driven programming model is that the actions of a program are scheduled strictly in the order of the occurrence of their associated events. Scheduling mechanisms for the concurrent execution of event-driven programs must not violate this property. A possible implementation would be to context-switch concurrent event-driven processes. Then, all the actions of a single program would still have run-to-completion semantics, while actions of separate processes could interrupt each other. However, because of the associated context-switching overhead of such a approach, system designers prefer to also schedule the actions of separate processes sequentially. For each event, the system software invokes the associated action of every process. Actions then always run to completion. This approach to scheduling actions of multiple, concurrent processes is depicted in Fig. 3.2.

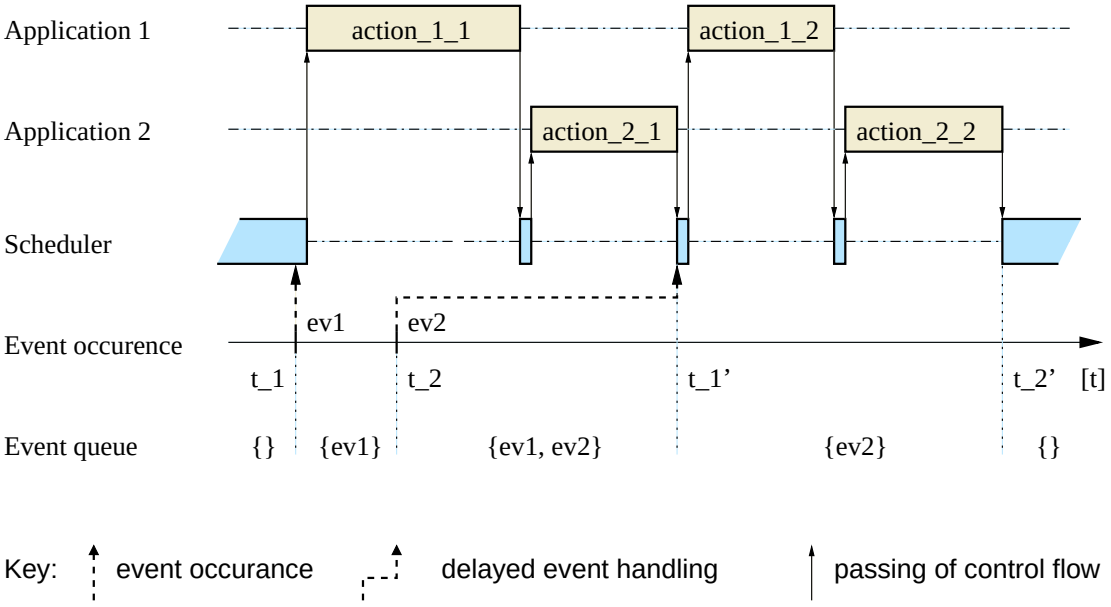



Figure 3.2: Scheduling concurrent, event-driven processes. For each event, the system software invokes the associated action of every process. Actions always run to completion; their execution order is arbitrary.

In Fig. 3.2 there are two event-driven programs running concurrently. Both of them react to events $ev_1$ and $ev_2$ (in $act_{x1}$ and $act_{x2}$, respectively), where $x$ denotes the application number. On the occurrence of $ev_1$ the system software first invokes $act_{11}$ (of application 1) then $act_{21}$ (of application 2). At time $t_1'$ all actions triggered by $ev_1$ have run to completion. Event $ev_2$ occurs before $t_1'$ and is stored in the event queue. Therefore, reactions to $ev_2$ are delayed until time $t_1'$, when it is dequeued.

Two questions become particularly apparent when studying Fig. 3.2. Firstly, in which order should the event handlers be invoked? And secondly, how long can the invocation of an action be delayed? The answer to the first question may be of interest when both programs modify a shared state, such as the state of system resources. Then, race conditions and possibly deadlocks may occur. Unless explicitly specified in the system documentation, application programmers

can not rely on any particular order as it depends on the actual implementation of the event-scheduling mechanism. In actual systems, the execution order of actions of separate processes is typically arbitrary.

The answer to the second question—how long can actions be delayed?— may be important when applications have real-time constraints. The answer depends on the execution times of actions from the different programs running concurrently and can therefore not be answered easily by any single application developer. From our experience with the event-driven model it is already hard to analyze the timing dependencies in a single application without appropriate real-time mechanisms. In a concurrent execution environment, where no single application programmer has the entire timing information, timing estimates are even harder to get by. Therefore, concurrent process models based on events can decrease the systems reliability. Since almost all application have at least some moderate real-time constraints, we believe that process concurrency should be avoided on event-driven sensor nodes.

Note that a question similar to the second (i.e., how long can the invocation of an action be delayed?) also arises in multi-treading environments. For such environments the reformulated question is: how much slower can the program become with multiple threads running concurrently. In multi-treading environments the execution delay of operations only depend on the own application code and the number of programs running concurrently, not the contents (i.e., the code) of other programs. For example, in the simple round-robin scheduling, the CPU time is equally divided among threads. If $n$ threads are running, each thread is allocated to the CPU for the length of a time slice before having to wait for another $n - 1$ time slices. Therefore the execution speed is only $1/(n)$-th of a system with only a single program executing. Actual implementations would be slower since context-switching consumes non-negligible time.

### 3.4.5 Analysis of Process Models

Mainly because of potentially unresolvable resource conflicts there are several reliability issues with process concurrency in sensor networks, regardless if its implementation is based on context-switches or events.

When appropriate mechanisms for inter-process protection are missing, which is the rule rather than the exception, a single deficient process can jeopardize the reliability of the entire system. Even without software bugs it is practically impossible to analyze in advance whether two concurrent processes will execute correctly. One problem is, that programs compete for resources in ways that cannot be anticipated by programmers. If one process requires access to exclusive resources, such as the radio or sensors, they may be locked by another process. Limited resources, such as memory and CPU time, may be unavailable in the required quantities. Identifying potential resource conflicts is very hard and fixing them with alternate control flows is practically infeasible. To make things worse, programmers of different applications typically cannot make arrangements on the timing of CPU intense computations and high resource usage. Therefore there are typically innumerable ways how two processes can ruin each others assumptions, particularly in the face of slow and resource-constrained sensor nodes.

Particularly the memory subsystem poses serious threads to concurrent programming. Sensor nodes do not provide virtual memory and therefore applications have to make do with the limited physical memory provided on a node. We expect that most non-trivial sensor-node programs occupy most of the systems' physical memory. It is generally hard to reach a good estimate of even a single program's memory requirements, particularly with dynamic memory allocation. Multiple competing programs certainly increase the uncertainty of such estimates. If processes require more than the available memory, system crashes or undefined behavior may result. As a consequence, the affected node or even an entire network region connected through that node may be unusable.

In sensor networks, concurrency on the process level does not play the same role as in general-purpose systems. Its purpose in traditional operating systems to increase processor utilization is contradictory to the design philosophy of sensor networks. In fact, as we pointed out earlier, most program architectures strive to maximize idle times in order to save power. Also, it is not expected that sensor nodes are used by multiple users to the same degree as traditional computing systems.

Though process concurrency may be a useful feature for sensor-network operators and users, we suggest its implementation only in sensor-node design with appropriate protection mechanism and resource provisions. In constrained nodes, however, the are severe reliability issues, which make its support prohibitive.

## 3.5 Overview and Examples of State-of-the-Art Operating Systems

In the previous sections we have discussed the principle programming and process models, as well as dynamic memory-management support of current sensor-node operating systems. The discussion represents the state-of-the-art of sensor-node programming and is based on current literature. The discussion focuses on resource-constrained sensor node platforms.

Before presenting two concrete examples of event-driven operating systems in greater detail, we will first summarize the previously discussed state-of-the-art sensor-node operating systems in Tab. 3.1. For each of the eight discussed operating systems, the table lists the system's programming language, the process and programming model, and the target-device features. The *concurrency* column under the heading *process model* denotes if the OS supports running multiple processes concurrently. The *dynamic loading* column under the same heading denotes if new processes can be loaded and started at runtime without also having to reload and restart the OS itself. Systems supporting dynamic loading of processes typically also support running multiple processes concurrently. The only exception is the Maté virtual machine, which supports dynamic loading of user applications but can only run one of them at a time.

Actual operating systems often implement variations of the basic programming models and often differ significantly in the system services provided. We will now present two representatives of current event-based operating systems; our BTnode system software, and the system that is sometimes referred to as the

| Programming Framework | | Process Model | | Programming Model | | Target Device |
|---|---|---|---|---|---|---|
| Operating System | Progr. Language | Con-cur-rency | Dy-namic Loading | Scheduling | Dyn. Mem. | processor core, clock, RAM, ROM, secondary storage |
| TinyOS [45] | nesC | no | no | events | no | Berkeley Motes 8-bit, 8 MHz, 4 Kb, 128 Kb, - |
| Maté VM [79] | Maté bytecode | no | yes | events | no | Berkeley Motes 8-bit, 8 MHz, 4 Kb, 128 Kb, - |
| BTnode [21] | C | no | no | events | no[a] | BTnode (ver. 1 and 2) 8-bit, 8 MHz, 4 Kb, 128 Kb |
| BTnut [123] | C | no | no | threads[b] | no[a] | BTnode (ver. 3) 8-bit, 8 MHz, 64 Kb, 128 Kb, 192 Kb RAM |
| SOS [52] | C | yes | yes | events | yes[c] | Berkeley Motes and others 8-bit, 8 MHz, 4 Kb, 128 Kb, - |
| Contiki [37] | C | yes | yes | events *and* threads[d] | no[a] | ESB node [47] 16-bit, 1 MHz, 2 Kb, 60 Kb, - |
| Impala [81] | n/a | n/a | n/a | prioritized events | yes[c] | ZebraNet node 16-bit, 8 Mhz, 2 Kb, 60 Kb, 512 Kb Flash RAM |
| MANTIS [8] | C | yes | yes | threads[d] | no | Mantis node 8-bit, 8 MHz, 4 Kb, 128 Kb, - |

[a] Dynamic memory allocation is provided by the standard C library libc for Atmel microcontrollers but is neither used in the OS implementation nor recommended for application programming.
[b] Cooperative multi-threading.
[c] Fixed (i.e., predefined) block sizes only.
[d] Preemptive multi-threading.

Table 3.1: Current programming frameworks for resource-constrained sensor nodes. The frameworks' runtime environment (as provided by the system software) supports different process models and programming models as discussed in sections 3.2 and 3.4. (n/a: feature not specified in the available literature.)

de facto standard of sensor-node operating systems, TinyOS. We will discuss their features and their deviations to the basic model in more detail.

### 3.5.1 The BTnode System Software

The BTnode system software (see Fig. 3.3) is a lightweight OS written in C and assembly language that has been initially developed for the first version of the BTnode. The system provides an event-based programming model.

It does not provide dynamic loading of processes and has a non-concurrent process model. Though dynamic memory allocation is available, the system software is only using static memory allocation to avoid reliability issues. Application programmers are also strongly discouraged to used dynamic memory allocation. The drivers, which are available for many hardware subsystems and extensions (e.g, the Bluetooth radio and several sensors subsystems), provide convenient, event-driven APIs for application development.

The BTnode system is composed of four principal components (see Fig. 3.3): the sensor-node hardware, the drivers, the dispatcher, and an application (which

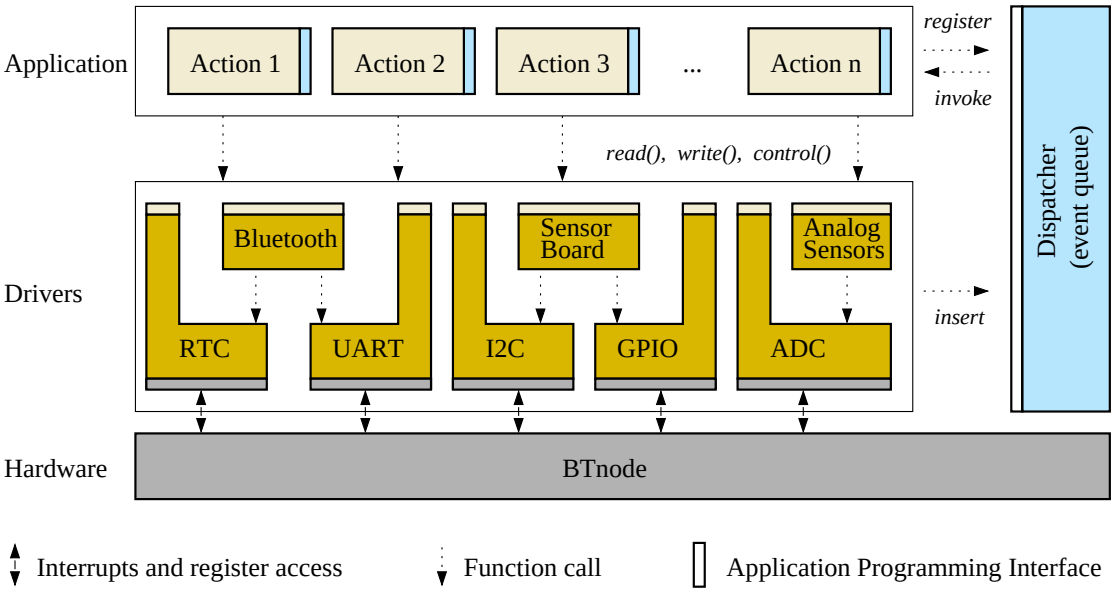



Figure 3.3: The BTnode system software and programming framework for WSN applications.

is composed of one or multiple actions). The components have clearly defined interfaces through which they are accessed. A central component is the dispatcher.

### Drivers and Dispatcher

The drivers have two interfaces, a lower interface to the hardware and an upper interface to the application code. Through the lower interface they receive and send IO data in a hardware-specific manner through interrupt service routines and hardware registers. Through their upper interface they provide a convenient user-programming API for controlling the hardware and IO. Drivers never call any application code directly. Instead, to notify the application of hardware state changes, they use the dispatcher's interface to insert an event into its event-queue. This dispatcher then invokes the appropriate application code (i.e., an action).

In the BTnode system software there is a hierarchy of drivers. Low-level drivers interact with the hardware directly (such the real-time clock driver). Higher-level drivers only interact with the hardware indirectly and require other lower-level drivers. For example, the Bluetooth driver relies on the real-time clock and the UART driver.

The drivers are designed with fixed buffer lengths that can be adjusted at compile time to meet the stringent memory requirements. Available drivers include nonvolatile memory, real-time clock, UART, $I^2C$, general purpose IO, LEDs, power modes, and AD converter. The driver for the Bluetooth radio provides a subset of the networking functionality according to the Bluetooth specification. It accesses the Bluetooth module through an UART at a speed of up to 230400 baud. Bluetooth link management is performed on Bluetooth's L2CAP layer. RFCOM, a serial port emulation, provides connectivity to computer terminals and consumer devices, such as cameras and mobile phones. Dial-up con-

nections to modem servers through a mobile GSM phone are easily established with a special-purpose function. All other GSM services (such as file sharing, phone book and calendar) can be utilized through lower-level interfaces.

### Programming Model

The system is geared towards the processing of (typically externally triggered) events, such as sensor readings or the reception of data packets on the Bluetooth radio. To this end, BTnode applications follow an event-based programming model, where the system schedules user-specified actions on the occurrence of events. A set of predefined event types is used by the drivers to indicate critical system conditions.

Upon detecting such a critical a condition, a driver inserts the corresponding event into the FIFO event-queue, which is maintained by the dispatcher. While the event-queue is not empty, the dispatcher removes the first event from the queue and invokes its associated user-defined action. Programmers implement actions as C functions. In actions, the programmer can use the (non-blocking) driver APIs to access IO data buffered in the drivers or to control their behavior. Programmers can also define their own event types and can insert instances of those events into the event queue from actions (not shown in Fig. 3.3). The different types of events are internally represented by 8-bit integer values (see Prog. 3.4 for examples).

Unlike several other event-based systems the association of events and actions is not fixed. Rather, applications can dynamically associate actions with event types. For this purpose, the dispatcher offerers a dedicated registration function, which takes the an event and an action as parameter (see `btn_disp_ev_reg()` in Prog. 3.4). Despite these dynamics, the association between events and actions typically remains relatively stable throughout the execution of a program. Often it is established only once during program initialization. After the dispatcher has been started, it accepts events from drivers and applications. It remains in control until an event is inserted into the queue. Then it invokes the corresponding action. Once the action is completed, the dispatcher checks for the next unprocessed event and invokes the corresponding action. Actions have a predefined signature as defined in Prog. 3.4, lines 11-13.

Actions have two arguments, called *call data* and the *callback data*. Both arguments are of an untyped field of fixed length (32 bit); they are passed to the action when it is invoked by the dispatcher. The *callback data* argument can be specified by programmers in the dispatcher's registration function. The argument is stored in the dispatcher and passed to actions when they are invoked. That is, the callback data is event-type specific. The *call data* argument is invocation specific. For user-defined event types the call-data values can be passed when inserting individual event instance into the queue. For system specified event types the call-data argument has a type-specific meaning. Its value is set by the driver generating the event. The predefined event types and their associated call-data arguments are:

- TIMEOUT_EV: a timeout has expired. The call-data parameter carries a timestamp to indicate when the timer was set to expire (as there may be scheduling delay).

Program 3.4: API of the dispatcher. Lines 1-4 show examples of predefined event definitions; in line 7-13 the signature of actions are defined; the dispatcher's registration function is shown in lines 16-19; the insert function is shown in lines 21-24.

```
1  #define UART0_RCV_EV        2   // data available for reading on UART0
2  #define BT_CONNECTION_EV    8   // Bluetooth connection occurred
3  #define BT_DISCONNECT_EV    9   // Bluetooth disconnect occurred
4  #define BT_DATA_RCV_EV     10   // Bluetooth data packet received
5  // ...
6
7  typedef uint32_t call_data_t;
8  typedef uint32_t cb_data_t;
9
10 // signature of an action
11 typedef void (*callback_t) (
12                 call_data_t call_data,
13                 cb_data_t   cb_data );
14
15 // the dispatcher's registration function
16 void btn_disp_ev_reg(
17         uint8_t     ev,             // event 'ev' triggers action 'cb'
18         callback_t  cb,             // with the parameter specified
19         cb_data_t   cb_data);       // in 'cb_data'
20
21 // the dispatcher's insert function
22 void btn_disp_put_event(            // insert 'ev' to the event queue
23         uint8_t ev,                 // 'call_data' is passed as argument
24         call_data_t call_data );    // to the action invoked by this event
```

- ADC_READY_EV: a data sample is available from the ADC converter and it is ready for the next conversion. The call-data carries the conversion value.

- I2C_RCV_EV: data has been received on the system's $I^2C$-bus and is now available for reading from the incoming buffer. The number of bytes available for reading at event-generation time is passed as call-data parameter.

- I2C_WRT_EV: the $I^2C$-bus is ready for transmitting data. The call-data parameter carries the current size of the outgoing buffer.

- UART_RCV_EV: data has been received on the system's UART and is now available for reading from the incoming buffer. The number of bytes available for reading at the event-generation time is passed as call-data parameter.

- UART_WRT_EV: the UART is ready for transmitting data. The call-data parameter carries the current size of the outgoing buffer.

- BT_CONNECTION_EV: a Bluetooth connection-establishment attempt has ended. The connections status (i.e., whether the attempt was successful or has failed, as well as the reason for its failure) and the connection handle are passed as call data.
- BT_DATA_RCV_EV: a Bluetooth data packet has been received and is ready for reading. The index to the packet buffer is passed as call data.

The code in Prog. 3.5 shows a typical BTnode program. The program waits until a Bluetooth connection is established remotely and then sends a locally sampled temperature value to the remote unit. During initialization (lines 5-11) the program registers the action `conn_action()` to be called on `BT_CONNECTION_EV` events (line 8) and `sensor_action()` to be called on `BT_ADC_READY_EV` events (line 10). It then passes control to the dispatcher (line 11), which enters sleep mode until events occur that need processing.

Once a remote connection is established, the Bluetooth driver generates the `BT_CONNECTION_EV` event and thus `conn_action()` (line 14) is invoked. In `conn_action()` the program first extracts the connection identifier from the call-data argument and saves to a global variable for later use (line 16). Then it starts the conversion of the analogue temperature value (line 18). On its completion, the `BT_ADC_READY_EV` event is generated and `sensor_action()` is invoked. Now the temperature value is extracted from the call-data argument (line 24) and sent back to the initiator (line 24), using the previously saved connection identifier. The same sequence of actions repeats on the next connection establishment. Until then, the dispatcher enters sleep mode.

### Process Model

Like most operating systems for sensor nodes, the BTnode system software does not support a dynamic process model. Only a single application is present on the system at a time. At compile time, applications are linked to the system software, which comes as a library. The resulting executable is then uploaded to the BTnode's Flash memory, effectively overwriting any previous application code. After uploading, the new application starts immediately.

However, the BTnode system can also be reprogrammed over-the-air using Bluetooth. To do so, the application currently running on the system needs to receive the new executable, save it to SRAM, and then reboot the system in bootloader mode. The bootloader finally transfers the received executable to Flash memory and starts it. Over-the-air programming is largely automated; it is a function of the system software but needs to be triggered by the user application.

### Portability

The whole system software is designed for portability and is available for different operation environments (x86 and iPAQ Linux, Cygwin, and Mac OS X) apart from the BTnode platform itself. These emulations simplifies application building and speed up debugging since developers can rely on the sophisticated debugging tools available on desktop systems. Also, the time for uploading the

Program 3.5: A simple BTnode program handling two events in two actions. The action `conn_action()` handles (remotely) established connections while `sensor_action()` handles (local) sensor values after they become available.

```
#include <btnode.h>
static uint16_t connection_id = 0;

int main( int argc, char* argv[] ) {
   btn_system_init( argc, argv, /* ... */ );
   btn_bt_psm_add( 101 ); /* accept remote connections */
   // register conn_action() to be invoked when a connection is established
   btn_disp_ev_reg( BT_CONNECTION_EV, conn_action, 0 );
   // register sensor_action() to be invoked when conversion is done
   btn_disp_ev_reg( BT_ADC_READY_EV, sensor_action, 0 );
   btn_disp_run();
   return 0; /* not reached */
}
void conn_action( call_data_t call_data, cb_data_t cb_data ) {
   // remember connection id for reply
   connection_id = (uint16_t)(call_data & 0xFFFF);
   // read temperature value from ADC converter
   bt_adc_start();
}
void sensor_action( call_data_t call_data, cb_data_t cb_data ) {
   // extract the converted value from the call-data argument
   uint8_t temperature = (uint8_t)(call_data & 0xFF);
   // reply with a packet containing only the temperature value (1 byte)
   btn_bt_data_send( connection_id, &temperature, 1 );
}
```

sensor-node application to the embedded target can be saved for testing. Furthermore, various device platforms (such as PCs or iPAQs running Linux) can be seamlessly integrated into BTnode networks (e.g., to be used as cluster heads), reusing much of the software written for the actual BTnode. These devices can then make use of the resources of the larger host platforms, for example, for interfacing with other wireless or wired networks, or for providing extended computation and storage services.

### 3.5.2 TinyOS and NesC

Another representative of an event-driven programming framework is the one provided by the TinyOS operating system and its incorporated programming language nesC [45, 59]. (Since operating system and programming language inherently belong together, we use the term TinyOS to denote both. Only where not clear from the context, we use TinyOS to denote the execution environment and operating system, and nesC to denote the programming framework and language.) TinyOS is often considered the de facto standard in WSN oper-

ating systems and programming frameworks today. Since its introduction in 2000, TinyOS has gained significant impact, possibly because of the success in the commercialization of its standard hardware platform, the Berkeley Motes. The early availability of the complementary combination of the freely available and well-supported programming framework and sensor-node hardware has made experimentation accessible to researchers without embedded systems background and without capabilities to develop and manufacture their own hardware.

TinyOS targets resource-constrained sensor nodes and has been implemented on the Berkeley Mote family as well as other sensor nodes, such as the BTnode [78]. Its goal is to provide standard sensor-node services in a modular and reusable fashion. TinyOS has a component-oriented architecture based on an event-driven kernel. System services (such as multi-hop routing, sensor access, and sensor aggregation) are implemented as a set of components with well-defined interfaces. Application programs are also implemented as a set of components. TinyOS components are programmed in a custom C dialect (called nesC), requiring a special compiler for development. A simple declarative language is used to connect these components in order to construct more complex components or entire applications. The designers of TinyOS consider it a *static language*, as it supports neither dynamic-memory allocation, nor process concurrency or dynamic loading. (It does, however, provide static over-the-air reprogramming).

TinyOS has an event-driven programming model. However, it has two levels of scheduling, rather than just one, as all other event-driven programming frameworks for sensor nodes. The scheduled entities are called *tasks* and *events*. This terminology is highly confusing: TinyOS *tasks* have no similarities to tasks in multi-tasking systems. Rather, a TinyOS task is what in event-driven systems is commonly known as an action or event handler. TinyOS tasks are scheduled sequentially by the scheduler and run to completion only with respect to other TinyOS tasks. They may be interrupted, however, by TinyOS events.

A TinyOS *event* is not what is commonly considered an event in the event-driven systems (i.e., a condition triggering an action). Rather, it is a time-critical computational function that may interrupt the regular control flow within TinyOS tasks as well as other TinyOS events. TinyOS events have been introduced so that a small amount of processing associated with hardware interrupts can be performed as fast as possible, without scheduling overhead but instead interrupting long-running tasks. TinyOS events are required to complete as fast as possible, in order not to delay any other time-critical TinyOS events. As such, TinyOS events are programming abstractions for interrupt service routines. Apart from being triggered by hardware interrupts, TinyOS events can also be invoked by regular code. Then they behave like regular functions.

Besides, TinyOS tasks and TinyOS events, TinyOS has a third programmatic entity, called *commands*. TinyOS commands resemble functions of sequential programming languages but have certain restrictions (e.g., they may not invoke TinyOS events).

TinyOS applications can have a graph-like structure of components but are often hierarchical. Each component has a well-defined interface that can be used to interconnect multiple components to construct more complex components

or to form an application. Each component interface defines the tasks, events, and commands it exports for use by other components. Each component has a *component frame*, which constitutes the component's context, in which all tasks, events, and commands execute and which stores its state. Components may be considered objects of a static object-oriented language. Components always exist for the entire duration of a process and cannot be instantiated or deleted at runtime. The variables within a component frame are static (i.e., they have global lifetime) and have component scope (i.e., are visible only from with in the component). However, multiple static instances of a component may exist as *parameterized interfaces*. Parameterized interfaces are statically declared as array of components and are accessed by index or name. The elements of a component array provide identical interfaces but have individual states. They are used to model identical resources, such as individual channels of a multi-channel ADC or individual network buffers in an array of buffers.

TinyOS is inherently event-driven and thus it shares the benefits of event-driven systems. As an event-driven system it has only a single context in which all program code executes. Therefore the implementation of TinyOS avoids context-switching overhead. All components share a single runtime stack while component frames are statically allocated, like global variables. As a deviation from the basic event-driven model, TinyOS has a two-level scheduling hierarchy which allows programmers to program interrupt service routines similar to event handlers using the TinyOS event abstraction. In most other event-driven systems, interrupts are typically hidden from the programmer and are handled within the drivers. IO data is typically buffered in the drivers and asynchronous events are used to communicate their availability to user code. In TinyOS, on the contrary, interrupts are not hidden from the programmer. This has been a conscious design decision. It allows to implement time critical operations within user code and provides greater flexibility for IO-data handling.

However, this approach has a severe drawback. TinyOS events may interrupt tasks and other events at any time. Thus TinyOS requires programmers to explicitly synchronize access to variables that are shared within tasks and events (or within multiple events) in order to avoid data races. TinyOS has many of the synchronization issues of preemptive multi-tasking, which we consider one of the main drawback of TinyOS. In oder to support the programmer with synchronization, more recent versions of nesC have an *atomic* statement to enforce atomic operation by disabling interrupt handling. Also, the nesC compiler has some logic to warn about potential data races. However, it cannot detect all race conditions and produces a high number of false positives. A code analysis of TinyOS (see [45]) and its applications has revealed that the code had many data races before race detection was implemented. In one example with race detection in place, 156 variables were found to have (possibly multiple) race conditions, 53 of which were false positives (i.e., about 30%).

The high number of actual race conditions in the TinyOS show that interruptible code is very hard to understand and manage and thus highly error prone. Even though the compiler in the recent version now warns about potential data races, it does not provide hints on how to resolve the conflicts. Conflict resolution is still left to programmers. The high rate of false positives (about 30%) may leave programmers in doubt whether they deal with an actual conflict or a false

positive. Chances are that actual races remain unfixed because they are taken for false alerts.

Also, as [30] has shown, programmers who are insecure with concurrent programming tend to overuse atomic sections in the hope to avoid conflicts. However, atomic sections (in user-written TinyOS tasks) delay the execution of interrupts and thus TinyOS events. As a consequence, user code affects–and may break!–the system's timing assumptions. Allowing atomic sections in user-code is contrary to the initial goal of being able to handle interrupts without scheduling delay. From our experience of implementing drivers for the BTnode system software we know that interrupt timing is highly delicate and should not be left to (possibly inexperienced) programmers but to expert system architects only.

## 3.6 Summary

In this chapter we have presented programming and runtime environments of sensor nodes. We have presented the three most important requirements for sensor-node system software: resource efficiency, reliability, and programming support for reactivity. We have also presented the three main programming models provided by state-of-the-art system software to support application programming. And we have presented common process models of sensor-node system software, which support normal operation of sensor nodes, that is, once they are deployed. Finally, we have presented selected state-of-the-art runtime environments for sensor nodes.

We have concluded that the control-loop programming model is overly simple and does not provide enough support even for experienced programmers. On the other hand, multi-threading is a powerful programming model. However, it has two main drawbacks. Firstly, multi-threading requires extensive system support for context switches. The per-thread data-structures (i.e., runtime stacks and switchframes), already consume large parts of a node's memory resources. Also, copying large amounts of context information requires many CPU cycles. Other authors before us have reached at the same conclusions (cf. [36, 37, 59, 60]). Also, multi-threading is considered too hard for most programmers because of the extensive synchronization required for accessing data shared among multiple threads. In [94] Ousterhout claims that multi-threading should be used only when the concurrent program parts are mostly independent of each other and thus require little synchronization. In sensor-node programming, however, threads are mainly used to specify the reactions to events that influence a common program state.

To provide a programming model without the need for processor sharing, per-thread stacks, and locking mechanisms, the event-driven programming model has been proposed for sensor-node programming. We and other authors have shown (both analytically and with prototype systems) that the event-driven programming model requires little runtime support by the underlying system software and can thus run on even the most constrained nodes. Yet it promises to be an intuitive model for the specification of sensor-node applications as it virtually embodies the event-action programming philosophy of wireless sensor networks.

Both of these models have their advantages and drawbacks, their followers and adversaries. In the context of sensor networks there is a bias towards event-based systems, mainly due to the existence of popular event-based programming toolkits such as TinyOS and nesC, which have a large user base. Hence, many sensor-network programmers are already used to think in terms of events and actions and find it natural to structure their programs according to this paradigm.

In the *next chapter* we will take a closer look at the event-driven model. We will analyze its expressiveness and resource requirements based on our own experience as well as on current literature. We will show that the event-driven model is also not without problems. There are also issues with reliability, program structure, and, ironically, with the memory efficiency of application programs (as opposed to operating systems, as is case with operating systems supporting multi-threading). We will show that these issues arise because the event-driven programming model is lacking a mechanism to clearly specify and memory-efficiently store temporary program states. But we will also sketch an approach to solve these problems. This approach will become the foundation of our OSM programming model and execution environment which we will present in chapter 5.

# 4 Event-driven Programming in Practice

The event-driven model is the favorite programming model among many sensor-network application programmers. In the last chapter we have shown several potential reasons for this preference: the model's event-action paradigm is intuitive and captures the reactive nature of sensor-node programs well, event-driven system software requires little of the node's constrained resources for its implementation, thus leaving most resources available for application programming, and finally, the early availability of the well-maintained event-driven TinyOS platform combined with a commercial sensor-node, the Berkeley Motes.

We have gained extensive experiences with the event-driven programming model ourselves. We have designed and implemented the BTnode system software, an event-driven runtime environment and programming framework for BTnodes (as presented in Sect. 3.5.1). Some of the system's more sophisticated drivers, such as the Bluetooth radio driver, are based on the event-driven model [21]. Also, we have developed many applications based on the event-driven model [21], mainly in the Smart-Its [62] and Terminodes [65] research projects.

However, in working with the BTnode system we have found two issues with the event-driven programming model. Firstly, the model is hard to manage, particularly as application complexity grows. Dunkles et al. [37] arrive at the same conclusion and [79] reports that programming TinyOS is "somewhat tricky" because of its event-driven programming model. And secondly, the event-driven programming imposes a programming style that results in memory inefficient application programs. Then, of course, two questions arise: What are the reasons for the model's drawbacks? And: Is there a fix? In this chapter we will give an answer to the first question and we will outline an answer for the second.

The baseline for both issues is that the event-driven model, though easy and intuitive, does not describe typical sensor-node applications very well. The model is sufficient for small sensor-network demonstrators and application prototypes. However, the model simply does not provide expressive enough abstractions to structure large and complex real-world sensor-network applications. Particularly, we believe that the model is lacking an abstraction to structure the program code along the time domain to describe program *states* or *phases*. The lack of such an abstraction is the fundamental reason for the above mentioned problems and is the starting point for developing our solution.

Phases are distinct periods of time during which the program (or part of it) performs a single logical operation. A phase may include several events, which are handled within the context of the logical operation. Sensor nodes need to perform a variety of ongoing operations that include several events. Examples

are as gathering and aggregating sensory data, setting up and maintaining the network, forwarding network packets, and so on. Typical sensor-node programs are structured in *distinct and discrete phases*, that is, the program's operations are clearly separable and execute mostly sequentially.

"Phase" is just one term that appears over and over in the WSN literature to describe the program structure of sensor-node applications and algorithms. Other frequently used terms with the same intention are "state", "role", "mode", or "part". Some authors even articulate that sensor-node programs in general are constructed as state machines [25, 37, 45, 80] because of their reactive nature. Notably, even multi-threaded programs are submitted to this view. Curiously, despite that fact that phases seem to be the prime concept in which application architects and programmers think, the predominant programming models for sensor nodes have no explicit abstraction to specify such phases. Only the authors of NesC have identified the lack of state support in [45] and have put their support on the agenda for future work.

We believe that the *state* abstraction of finite state machines is an adequate abstraction to model sensor-node program phases and that a programming model based on finite state machines can eventually solve the problems of the event-driven model. Particularly, we propose a programming model based on concurrent and hierarchical state machines, as proposed in the seminal work on Statecharts [53] by Harel.

In this chapter we will first examine the problem symptoms of the event-driven model in Sect. 4.1. Then, in Sect. 4.2, we will present what we call the anatomy of sensor-node programs, that is, typical and recurring structures of phases in sensor-node programs and their interaction. We will explain these structural elements step-by-step based on a small example program, which is commonly found within the WSN literature. To illustrate this program's structure we will use a small subset of the graphical Statechart notation. Alongside the presentation of the anatomy, we will show the difficulties in expressing its structural elements in the event-driven model. And we will point out how these difficulties lead to the symptoms laid out in the first section of this chapter. In the last section of this chapter, Sect. 4.3, we will sketch the basic idea of how to solve the problems of the event-driven programming model for the specification of sensor-node programs. We propose an extended programming model, which combines elements of the event-driven programming model with those of finite state machines. A concrete solution will be presented in the next chapter.

## 4.1 Limitations of Event-Driven Programming

In our work we have identified two issues with the otherwise very intuitive event-based programming model. These issues, which we call *manual stack management* and *manual state management*, arise because in the event-driven model many conceptual operations need to be split among multiple actions. In the following we will detail these issues and their causes.

Since actions must not monopolize the CPU for any significant time, operations need to be non-blocking. Therefore, at any point in the control flow where an operation needs to wait for some event to occur, the operation must be split

into two parts: a non-blocking operation request and an asynchronous completion event. The completion event then triggers an action that continues the operation. As a consequence, even a seemingly simple operation can lead to *event cascades* – an action calls a non-blocking operation, which causes an event to occur, which, in turn, triggers another action. Breaking a single conceptual operation across several actions also breaks the operation's control flow and its local variable stack.

Breaking conceptual operations among multiple functions has two implications for the programmer. Firstly, as the stack is unrolled after every action the programming language's variable-scoping features are effectively discarded. Programmers cannot make use of local, automatically managed variables but instead need to manually manage the operation's variable stack. This is called manual stack management [9]. Secondly, programmers must guarantee that *any order* of events is handled appropriately in the corresponding actions. This requires the manual intervention of the programmer by writing extra code. We call this manual state management. We use Program 4.1 as an example to clarify these issues.

Program 4.1: Event-driven code fragment to compute the temperature average of sensor nodes in the one-hop vicinity.

```
1  int sum = 0;
2  int num = 0;
3  bool sampling_active=FALSE;
4
5  void init_remote_average() {
6     sampling_active=TRUE;
7     sum = num = 0;
8     request_remote_temp();
9     register_timeout( 5 );
10 }
11  void message_hdl( MSG msg ) {
12    if( sampling_active == FALSE ) return;
13    sum = sum + msg.value;
14    num++;
15 }
16 void timeout_hdl() {
17    sampling_active=FALSE;
18    int average = sum / num;
19    /* ... */
20 }
```

Program 4.1 calculates the average temperature of sensor nodes in a one-hop distance. To do so, the sensor node running the program sends a broadcast message to request the temperature value from all neighboring sensor nodes (line 8). It then collects the remote samples in the event-handler for incoming radio messages (lines 11-15). A timeout is used to ensure the temporal contiguity of remote sensor readings. Finally, when the timeout expires, the average remote

temperature is calculated (line 18). As shown in the code above, this relatively simple operation needs to be split into three parts: 1) sending the request, 2) receiving the replies, and 3) processing the result after the timeout.

### 4.1.1 Manual Stack Management

In the above example, the data variables `sum` and `num` are accessed within two actions. Therefore `sum` and `num` cannot be local variables of either function. Instead they are declared as global variables, which have global scope and global lifetime.

In a traditional, purely procedural program, local variables serve the purpose of keeping an operation's local data. They are automatically allocated on the local runtime stack upon entering a function and are released on its exit. However, automatic variables cannot be used for event cascades since the local stack is unrolled after the execution of every action. Therefore, the state does not persist over the duration of the whole operation. Instead, programmers must manually program how to retain the operation's state. They can do so either using global variables (as in the example above) or by programming a state structure stored on the heap.

Both approaches have drawbacks. The global variables have global lifetime, that is, they permanently lock up memory, also when the operation is not running. For example, a global variable used for a short phase during system initialization resides in memory for the rest of the program's execution. Manually reusing this memory in different program phases is possible but highly error-prone. The (possibly multiple) programmers would have to keep track of the use of every variable's memory in each program phase. Besides their global lifetime, global variables also have global scope. That is, their visibility extends to the entire program. As a consequence, global variables can be read and modified throughout the entire program. To avoid naming conflicts and accidental access, strict naming conventions have to be introduced and followed by programmers. For these reasons, it is generally considered good programming practice to avoid global variables and use local variables instead. Local variables have been one of the achievements of structured programming.

The second approach, managing the operation's state on the heap, requires manual memory management (e.g., by using `malloc()` and `free()`), which is generally considered error-prone. It also requires system support for dynamic memory management, which has a significant resource penalty and is therefore sometimes missing for resource-constrained sensor nodes. We have discussed dynamic memory management for sensor nodes in the previous Sect. 3.3.

### 4.1.2 Manual State Management

Depending on the node's state and history of events, a program may need to (and typically does) behave quite differently in reaction to a certain event. In other words, the actual reaction to an event not only depends on the event but also on its context. This multiplexing between behaviors must be implemented explicitly by the programmer. As a result, programmers must include additional

management code, which obscures the program logic and is an additional source of error.

In the previous Program 4.1, for example, replies from remote sensors should only be regarded until the timeout expires and thus the timeout action is invoked. After the timeout event, no more changes to `sum` and `num` should be made (even though this is not critical in the concrete example). To achieve this behavior, the timeout action needs to communicate with the radio-message action so that no more replies should be regarded. Manual state management is highlighted in the program code. The desired behavior is enforced by introducing a (global) boolean flag `sampling_active` (line 3), which is set to indicate the state of the aggregation operation. The flag is used in all three functions (lines 6, 12, and 17). In the message action, the program checks whether the timeout has occurred already and thus remote temperature values should no longer regarded (line 12).

In general, programmers using the event-driven programming model must manually manage the program state in order to be able to decide in which context an action has been invoked. Based on that state information programmers must multiplex the program's control flow to the appropriate part of the action. We call this manual state management. Coding the flow control manually requires operating on state that is shared between multiple functions (such as the flag `sampling_active` in our example). Again, this state needs to be managed manually by the programmers. A general pattern for a well-structured implementation of context-sensitive event handling is shown in Program 4.2.

### Accidental Concurrency

Manually managing the program state may lead to accidentally (re)starting an event cascade, which we call *accidental concurrency*. In our example, the initialization function `init_remote_average()` and the two actions (`message_hdl()` and `timeout_hdl()`) implement a single logical operation that runs for some period of time (i.e., the aggregation phase). The parts of that operation belong inherently together and it is implicitly expected that the entire phase has been completed before it is started anew. However, our example is prone to accidental concurrency. The event cascade implementing the aggregation phase could be started anew with a call to `init_remote_average()` (perhaps from an action handling incoming network packets for re-tasking the node). Restarting the aggregation phase while a previously started aggregation phase is still running will lead to unexpected behavior and thus impair the reliability of the program. In our example, the global variables `sum` and `num` will be reset to zero, yielding an incorrect result for the previous instantiation of the remote-average operation. The topic of accidential concurrency is closly related to reentrant functions in multithreaded systems. Just like functions, most event cascades are not reentrant when they rely on global variables to store "private" state. In event-driven programming, however, global variables need to be used often, as explained in the previous section.

Accidential concurrency is not limited to multiple instances of the same cascade of events. Some locical operations (i.e., event cascades) may be mutually exclusive or may need to be executed in a particular order. Events that occur

Program 4.2: General pattern to implement context sensitive actions.

```
1  typedef enum {INIT, A, B, ... } state_t;
2  state_t state = INIT;                          // initial state
3
4  void event_handler_x {
5     switch( state ) {
6        case INIT:
7           // handle event_x in state INIT
8           state = ...;                          // set new state
9           break;
10       case A:
11          // handle event_x in state A
12          state = ...;                          // set new state
13          break;
14       case B:
15          // handle event_x in state B
16          state = ...;                          // set new state
17          break;
18       default:
19          // ...
20    }
21 }
22 void event_handler_y {
23   // ...
24 }
```

when the programmer was not expecting them (e.g., duplicated network messages or events generated by another action), however, will trigger their associated operation regardless. To enforce the correct behavior, programmers need to manage contex information and need to code the program's flow control explicitly.

In the event-driven model the reaction to an event cannot be specified with respect to such context information. Instead an event always triggers its single associated action. The event model does not provide any means against restarting of such phases anew. (In fact, there is not even an abstraction to indicate which parts of the code implement a single logical operation.) Special care must be taken by programmers not to accidentally restart a cascade of actions. In oder to be able to detect such errors, a defensive programmer would set a flag at the beginning of the logical operation and check it in every action to trigger exception handling. For the example Program 4.1, this could be achieved by checking whether every invocation of `init_remote_average()`) has been followed by a call to `timeout_hdl()` or otherwise ignore the invocation. The following additional code implements a possible solution when inserted after line 5 of the example:

```
if( sampling_active == TRUE ) {
  return; // error, ignore
}
```

#### Impaired Modularity and Extensibility

Now consider adding a new feature to the system, for example, for re-tasking the node. Say, the new subsystem interprets a special network-message event, which allows to configure which type of sensor to sample. Thus, the new subsystem requires a message handler, such as the one in line 11 of Program 4.1. Since each type of event always triggers the same action, both the sensing subsystem and the new re-tasking subsystem need to share the single action for handling network messages (`message_hdl()` in the example). The code fragment in Program 4.3 shows a possible implementation. Generally, extending the functionality of a system requires modifying existing and perfectly good code in actions that are shared among subsystems. Having to share actions between subsystems clearly impairs program modularity and extensibility.

Program 4.3: An action shared between two subsystems of a node (for aggregating remote sensor values and re-tasking).

```
1  void message_hdl( MSG msg ) {
2    if( msg.type == RETASKING )
3      // re-tasking the node
4      // ...
5    else {
6      // aggregating remote values
7      if( sampling_active == FALSE ) return;
8      sum = sum + msg.value;
9      num++;
10   }
11 }
```

### 4.1.3 Summary

In the small toy example we have just presented, manual state and manual stack management seems to be a minor annoyance rather than a hard problem. However, even in such a simple program as Prog. 3.5, a significant part of the code (4 lines plus one to avoid accidental concurrency) is dedicated to manual flow control. Even minor extensions cause a drastic increase in the required management code. As application complexity grows, these issues become more and more difficult to handle. In fact, in our applications that implement complex networking protocols (e.g., our Bluetooth stack), significant parts of the code are dedicated to manual state and stack management. The code is characterized by a multitude of global variables, and by additional code in actions to manage the program flow. This code obscures the program logic, hampers the program's readability, and is an additional source of error.

In general, the event-driven programming model implies to structure the program code into actions. Actions typically crosscut several logical operations (i.e., phases or subsystems), which hampers program modularity and extensibility. If functionality is added to or removed from the software, several actions and

thus subsystems are affected. Also, the event-driven programming model lacks methods to hide the private data of program phases from other phases, which also negatively effects the program structure and modularity. Finally, because local variables cannot be used across multiple actions, the model makes it hard to reuse memory, which results in memory inefficient programs. These symptoms of event-driven programs are by no means restricted to our event-based programming framework for BTnodes. Similar issues can be found in program examples for the popular TinyOS / NesC framework in [45] or for SOS in [52].

## 4.2 The Anatomy of Sensor-Node Programs

Most sensor-network programs and algorithms are structured along discrete phases. An example where the composition of sensor-node programs into distinct, sequentially scheduled phases becomes particularly apparent is role assignment: The node only performs the single currently assigned role out of several possible roles (see for example [42, 56, 98]. When a new role needs to be assigned, the previous role is first terminated before the newly assigned role is started.

In fact, a prime application of sensor networks is to detect certain states of the real world, that is, phases during which a certain environmental condition is met. In applications where the state detection is performed partly or entirely on the sensor nodes (as opposed to in the background infrastructure based on the information collected by sensor nodes) these real-world states are often modeled discretely for the sake of simplicity. Detecting a change in the real-world state then typically triggers a new phase in the sensor-node program in order to react to the change in a state-specific manner. A typical reaction is to collect additional information specific to the detected state and to notify the rest of the sensor network of the detection. This phase continues until the real-world state changes again, which is reflected by yet another phase change of the program. Examples are a health monitor, which determines the stress state of a human user to be high, normal, or low [55], an augmented mobile phone, which automatically adapts it ring-tone-profile based on the detected phone context (in-hand, on-table, in-pocket, and outdoors) [46], several tracking applications (tracking target is detected or absent) [6, 100, 118], product monitoring (product damaged or intact) [107], and cattle herding (animal is inside or outside of a virtually fenced in region) [28].

### 4.2.1 Characteristics of a Phase

We have analyzed the source code of numerous programs and program fragments (both from the literature as well as our own) if and how program phases are reflected in the code. We have found that phases can indeed be identified in program sources. The phases of programs and algorithms do share four common characteristics by which they can be recognized: (1) they run for a certain period of time, during which they may need to handle several events, (2) their beginning and end are typically marked by (one or several) specific events, (3) they are strongly coupled, that is, the computational operations within a phase

utilize a common and clearly defined set of resources, and (4) they start with an initialization function where the (temporary) resources for the phase are allocated and they end with a deinitialization function where the resources are released again.

Individual phases have a strong internal coupling with respect to other phases. Typically this means two things: Firstly, the computations belonging to a phase operate on a common data structure that is private with respect to other phases (though phases do typically communicate via small amounts of shared data). And secondly, during a phase the program utilizes a well defined set of resources, which may differ significantly from other phases. For example, phases differ in the amount of computation taking place (CPU cycles), the size of the data structures being used (memory), which sensors are being sampled and at what rates, and whether the radio is actively sending, in receive mode, or switched of entirely.

In the program code, the start of a phase is typically marked with an initialization function and ends with a deinitialization function (cf. [52]). In initialization, the resources required by the phase are allocated and its data structures are set to their initial values. Resource allocation can mean that memory for holding private data structures is allocated, that timers are set up, and that sensors or the radio are switched on. At the end of a phase, information computed during the phase may be made available for use by other (i.e., concurrent or subsequent) phases, for example, by saving it into a shared data structure. Then the phase's resources are released.

### 4.2.2 Sequential Phase Structures

In the previous section we have described the characteristics of an individual phase. We will now describe the phase structure of sensor-node programs, that is, typical and recurring patterns of how programs are constructed of individual phases. This description is based on the following running example. Consider a monitoring application that samples an environmental parameter (e.g., the light intensity) at regular intervals and then immediately sends the data sample to a dedicated sensor node. This simple application (sometimes named "Surge", "sense and forward" or "sample and send") is used with minor modifications throughout the sensor-network literature to demonstrate programming-framework features [8, 21, 45, 48, 52, 79] or to do performance evaluations [36, 48, 79]. We will use this example and modifications of it to describe the phase structure of sensor-node programs. As a matter of fact, this application is so simple that the code fragments found in the literature typically treat the whole program as a single entity with a single data structure and all initialization performed at the start of the program. Yet, we will gradually add features and treat it like a full featured sensor-node program to explain program structures with multiple phases and their interactions.

> ***Surge Example 1***
> A simple version of Surge consists of four phases, which we will call INIT, SAMPLE, SEND, and IDLE, as depicted in Fig. 4.1. The program starts with the INIT phase, which performs the general system initialization. When the INIT phase is complete, the program enters the

SAMPLE phase in order to start sampling. In order to save power the sensor is generally switched off unless actually sampling. Therefore the SAMPLE phase is initialized by switching on the sensor. When the data sample is available, which may considerable time (e.g., for GPS sensors), the sensor is switched off again. Then the SEND phase is started. Similar to the SAMPLE phase the SEND phase first switches on the radio, waits until a connection to the dedicated sink node is established and then sends the data sample acquired in the previous SAMPLE phase. When the reception of the data has been acknowledged by the sink node, the radio is switched off again and the program moves to the IDLE phase. In this initial version of the program we ignore a failed send attempt (e.g, due to collisions or packet loss on a noisy channel) and move to IDLE anyway. The IDLE phase does nothing but wait for the start of the next interval, when the next cycle of sampling and sending is restarted, that is, the program moves to the SAMPLE phase again. In order to sleep for the specified time, a timer is initialized upon entering the IDLE phase and then the CPU is put into sleep mode. The node wakes up again when the timer fires and the program moves to the SAMPLE mode again. In our example, the sampling of the sensor in the SAMPLE state may need to be scheduled regularly. Since both the sample and the send operation may take varying amounts of time, the program needs to adjust the dwell period in the IDLE phase. □

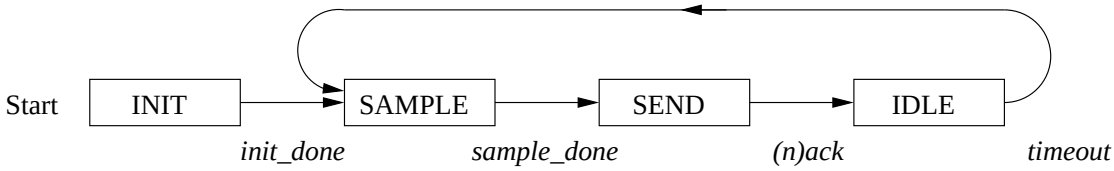



Figure 4.1: A simple version of Surge with four program phases. In this version a failure while sending the data sample is ignored.

Though this program is rather simple, it has many features of the complex algorithms used in current sensor-node programs.

### Sequential Composition

The four phases of the Surge program are all scheduled strictly sequentially. From our analysis of sensor-node programs we conclude that typically the majority of logical operations (i.e., phases) are performed sequentially (though there is a certain amount of concurrency, as we will discuss in Sect. 4.2.3).

Whenever the application at hand permits, architects of sensor-node programs seem to be inclined to schedule these phases sequentially rather then concurrently. This holds true even for many fundamental system services, such as network management, routing, device management, etc. This is in contrast to traditional networks, where most of such services are performed permanently in the background, often by dedicated programs or even on dedicated hardware.

We think that the composition of sensor-node programs predominantly into distinct, sequential phases has two main reasons: Firstly, sequential programs are typically easier to master than concurrent programs, as they require little explicit synchronization. And secondly, and even more importantly, the phases of a sequential program do not require coordination of resources and can thus utilize the node's limited resources to the full, rather than having to share (and synchronize) between concurrent program parts.

One difficulty of distinct, sequential program phases in the event-driven model lies in the specification and the memory efficient storage of variables used only within a phase. As we discussed earlier, such temporary variables must typically be stored as global variables, with global scopes and lifetimes. From the program code it is not clearly recognizable when a phase ends and thus the variables' memory can be reused.

### Starting and Completing Phases

In sensor-network programs a phase often ends with the occurrence of a particular event, which then also starts the next phase. Often this event is an indication that a previously triggered operation has completed, such as a timeout event triggered by a timer set previously, an acknowledgment for a network message, or the result of a sensing operation. So it can often be anticipated *that* a particular event will occur, is often hard or impossible to predict *when*.

A phase may also end simply because its computation has terminated, like the INIT phase in the previous example. Then the next operation starts immediately after the computation. It may still be useful to view both phases as distinct entities, particularly if one of them can be (re-)entered along a different event path, as in the above example.

The specification of phase transitions in the event-driven programming model poses the aforementioned modularity issue. Since the transition is typically triggered by a single event, both the initialization and deinitialization functions need to be performed within the single associated event handler. Therefore, modifications to the program's phase structure causes modifications to all event handlers that lead in and out of added and removed phases.

Also, phase transitions incur manual state management if different phases are triggered by the same event. Then the associated event handler must be able to determine which phase's initialization to perform. This can only be done by manually keeping the current program state in a variable and checking the previous state at the beginning of each action.

### Repeated Rounds of Phases

Often phases of sensor-node programs are structured in repeated rounds. A typical example is the cyclic "sample, send and sleep" pattern found in our example application, as well as in many sensor-node programs described in the literature. Examples where repeated rounds of distinct, sequentially scheduled program phases are used are LEACH [55, 56], TinyDB and TAG [83, 84], and many more.

### Choosing the next phase: Branching

Often there are several potential next phases in sensor-network programs. The choice of the actual next phase then depends on an input event, the state of a previous computation, or a combination of both. Also, a single phase may be reachable through multiple paths, as in the following example.

> ***Surge Example 2***
> Until now our Surge application ignored a failed sending attempt. If sending fails, however, the program may decide to store the data sample in a local buffer (rather then ignoring the failure). The modified program could then, for example, send the stored data sample in the next round together with the new sample. An updated version of the program is depicted in Fig. 4.2. In the new version, the choice of the next phase from SEND depends on the event generated by the send operation (*nack* or *ack*). Likewise, the IDLE phase may be reached from two phases, namely SEND and STORE. □

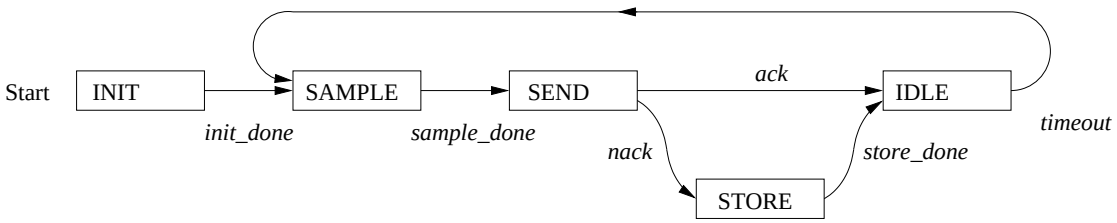



Figure 4.2: A more sophisticated version of Surge with five program phases. In this program version a failure while sending the data sample is handled by buffering the data locally. Note that the control flow branches from the SEND phase depending on the events *nack* and *ack*.

### Preemption

In sensor-node programs the occurrence of a certain event often interrupts (we say it *preempts*) a sequence of multiple phases in order to start a new phase. That is, if the event occurs in any phase of a sequence of phases, the control flow of the program is transfered to the new phase.

> ***Surge Example 3***
> Let us now consider yet another variation of the basic Surge program, which is inspired by the modification of Surge as distributed with TinyOS [125]. In this version we introduce a network-command interface that allows a remote user to put the node in sleep mode by sending a *sleep* message to it. The node then preempts all current activities and sleeps until it receives a subsequent *wakeup* message. This version of Surge is depicted in Fig. 4.3. □

The dashed lines in Fig. 4.3 denote the preemption caused by the *sleep* event. That is, the control is passed to the SLEEP phase from any of the phases SAMPLE, SEND, and IDLE on the occurrence of the *sleep* event.

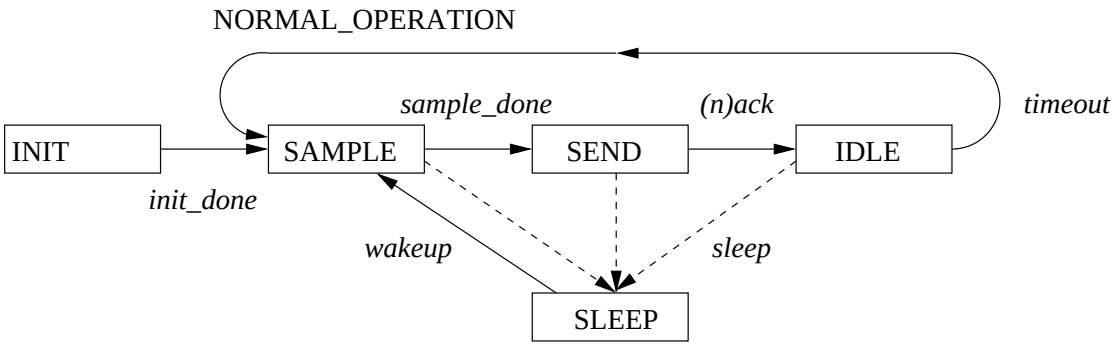



Figure 4.3: The three program phases SAMPLE, SEND, and IDLE are preempted by the $sleep$ event.

The execution of the target phase of a preemption takes priority over the preempted phases. Preemption is basically the same as what we have previously called branching, only that preemption has a number of source phases (rather then just one). Preemption is can be considered as handling of exceptions. The phase to which control is transferred to may then be considered the exception handler.

Specifying preemption in the event-model requires extensive state management. The difficulty is two-fold: Firstly, on the occurrence of a preempting event, the programmer must first deinitialize the phases that are to be preempted. Special care must be taken if preemption is added during the redesign of a sensor-node program. From our experience, proper deinitialization can be easily forgotten. And secondly, (future) events that are caused by computational operations performed before the preemption occurred must be actively ignored. For example, in our example program the $(n)ack$ event is caused by sending a data sample. If the send operation has completed, but a $sleep$ event is received next (i.e., before the $(n)ack$ event), the $(n)ack$ event is nevertheless generated by the OS and thus triggers the associated action. In this action, however, it needs to be ignored. The code fragment in Prog. 4.4 shows a possible implementation of the preemption in the event-driven model. The handler of the $sleep$ event (lines 1-14) implements the preemption. The preempted phases are deinitialized within the switch statement (lines 3-11). The handler of the $nack$ event first checks the exceptional case when the program is in the SLEEP state (lines 16). Then the $nack$ event is ignored. If the program was coming from the SEND phase (which is the regular case), it behaves as without preemption. Clearly, the complexity of manual state management in the case of preemption is hard to handle for programmers. Even this short program is hard to write and even harder to read. To deduce a programmers intention from the code is next to impossible.

### Structuring Program Phases: Phase Hierarchies

In some cases, it is convenient to think of a set of sequentially composed phases as a single, higher-level program phase. This implies that there is a hierarchy of phases, where phases higher in the hierarchy (which we call super-phases) are composed of one or more sub-phases. Our previous example, Fig. 4.3, is such a case.

Program 4.4: A code fragment showing the full deinitialization procedure that needs to be performed on a preemption.

```
1  void sleep_hdl() {
2   // deinitialize previous phases
3    switch( sate ) {
4      case SAMPLE: ...   // deinitialize SAMPLE phase: switch off sensor
5        break;
6      case SEND:   ...   // deinitialize SEND phase: switch off radio
7        break;
8      case IDLE:   ...   // deinitialize IDLE phase: reset timer
9        break;
10     default:     ...   // error handling
11   }
12   state = SLEEP;
13   // do sleep ...
14 }
15 void nack_hdl() {
16   if( state == SLEEP ) {
17      // ignore the nack event while in SLEEP
18      return;
19   }
20   // deinitialize the SEND phase: switch off radio
21   state = IDLE;
22   // perform initialization and operations for IDLE
23 }
```

The complete cycle of sampling, sending, and idling can be considered a phase by itself, representing the period where the node performs its normal operation. The new situation is depicted in Fig. 4.4, where we have indicated the super-phase by surrounding the contained phases with a solid box. The new high-level phase NORMAL_OPERATION is initially entered by the *init_done* event coming from INIT phase. It may be re-started by the *wakeup* event after residing temporarily in the SLEEP phase. NORMAL_OPERATION ends on the occurrence of the *sleep* event. Then the high-level phase as well as all contained phases are preempted and control is passed to the SLEEP phase.

Super-phases are a convenient mind model that allows to subsume a complex subsystem into a single, less-refined subsystem. Hierarchical phase structures help program architects to develop a clear and easy to understand model of the program. These structures also foster communication about the model on different levels of detail. It is easy to "abstract away" the details of a super-phase (top-down view, see Fig. 4.5) or, on the contrary, to first focus only on the details and building the system from the ground up (bottom-up view).

Super-phases typically have the same characteristics as uncomposed phases. That is, the entire super-phase may have its own initialization and deinitialization function, as well as data structures. Then, all sub-phases (i.e., all composed phases) share that common data structure. The super-phases start when a contained phase starts and ends when the last of the contained phases is left.

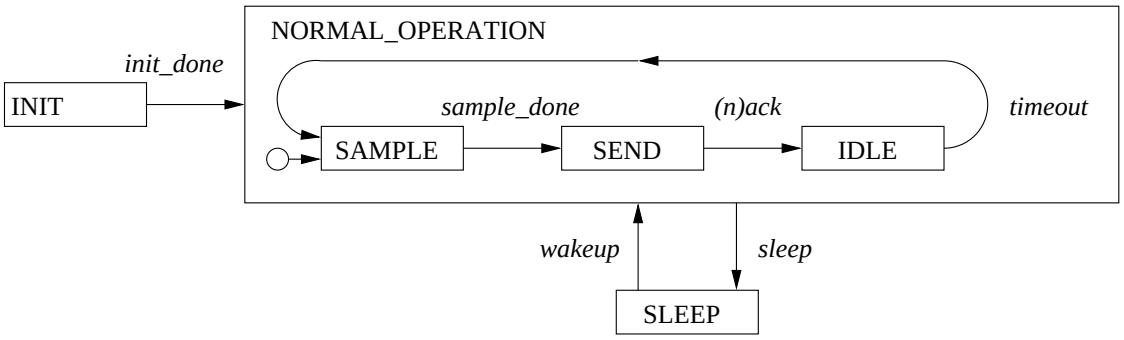



Figure 4.4: A refinement of Fig. 4.3. The three phases SAMPLE, SEND, and IDLE have been combined into the super-phase NORMAL_OPERATION. The super-phase is preempted by the *sleep* event.

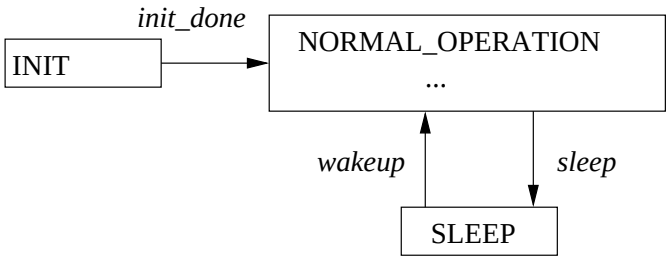



Figure 4.5: An abstraction of NORMAL_OPERATION from Fig. 4.4.

There may be a dedicated initial phase when starting the composed phase, as the SAMPLE phase in our example (indicated by the arrow origination from a circle). However, there may be several initial states, where the actual initial state depends on the result of a previous computation, the event that caused the start of the composed phase, or the phase from which control was passed to the composed phase. The problem of implementing hierarchical phases in the event-driven model is to correctly perform the (de)initialization functions correctly and to track resource usage within sub-phases.

In the sensor-network literature there are several examples where authors describe their algorithms as hierarchical phases. The LEACH algorithm [56] (a clustering-based communication protocol to evenly distribute the energy load among the sensors in the network), for example, is described as being "broken into rounds, where each round begins with a set-up phase [...] followed by a steady-state phase". The set-up phase is further composed of three sub-phases called *advertisement*, *cluster set-up*, and *schedule creation*.

### Phase-Bound Lifetime and Scope of Data Structures

Sensor-node programs often operate on data structures that are temporary in nature. That is, the use of such data structures (just like other resources) is typically closely tied to a phase or a sequence of multiple phases; the structures are accessed only during certain phases of the program. Typically, programs use several data structures in different phases, that is, they have different lifetimes. A challenge in programming sensor nodes is the efficient management of data structures, since memory is one of the highly-constraint resources. That means

that programmers typically strive to constrain the lifetime of data structures to those program phases in which they are actually needed. To summarize, the lifetime of temporary data structures *must* extend to all phases that share the data, but *should* not extend beyond those phases. That is, after the data structures are no longer needed, the memory should be released in order to recycle the memory in distinct phases.

In our example, the SAMPLE and SEND phases both need access to the sampled data, which may have an arbitrary complex in-memory representation depending on the sensor and application. Therefore, the lifetime of the data sample must extend to both the SAMPLE and SEND phases. After sending the value the data becomes obsolete. The memory for the data could be reused in subsequent phases. Since in our example the IDLE phase does not have extensive memory requirements, there would be not much point. In real-world sensor-node programs, however, memory reuse is imperative.

Again, in LEACH [56], for example, individual phases with seizable memory requirements alternate. The operation of each sensor node participating in the LEACH algorithm is broken up into rounds, where each round begins with a cluster-organization phase, followed by a data-aggregation, and data-forwarding phase. In the cluster-organization phase each node collects a list of all network neighbors annotated with various information, such as received-signal strength. In a dense network, this list can be of considerable size. During the phase, each node chooses a cluster head and may become one itself. After completion of the phase, only the identity of the cluster head is required, the neighbor list however has become obsolete and can be freed. The outcome of the cluster-organization phase is shared with the rest of the program by means of shared memory. Every node that has become a cluster head then starts collecting data samples from their slaves, aggregates the received data, and finally sends the aggregate to its own cluster head, whose identity has been kept from the cluster-organization phase. Here, releasing the temporary data structure (i.e., the neighbor list) frees up memory that might be required during data collection and aggregation.

In the event-driven programming model programmers have two alternatives to store such temporary data: global variables and dynamic memory. As we have previously shown in Sect. 4.1, both approaches have severe drawbacks.

### 4.2.3 Concurrency

While sensor-node programs are often geared towards sequential operation, it may still be necessary or convenient to run some operations concurrently. Probably the most common concurrent activities performed by sensor-node programs are sampling and maintenance tasks, such as network management. For example, in EnviroTrack [6], a group-based protocol for tracking mobile targets, sensing runs concurrently to group management.

As we pointed out in a previous chapter, in sensor-node programming, concurrency is typically a purely logical concept because sensor nodes are single-processor systems. Concurrency is a descriptive facility for the programmer to describe self-contained and concurrently executing parts of the system in individual programmatic entities, without having to care about their actual execu-

tion. The system software takes care of scheduling these entities so that they appear to run concurrently, while they are actually scheduled sequentially on the single processor in time multiplex.

The event-driven programming model has no notion of concurrency, such as the thread abstraction in multi-threaded programming. Indeed, it does not even have a programming abstraction to group together the logical operations and self-contained system parts. As we have discussed in the previous sections, despite the lack of a dedicated programming abstraction, sensor-node programmers tend to think and structure their programs in terms of hierarchical phases. The phase model lends itself well to modeling concurrent program parts.

***Example Surge 4***

Let us consider a final addition to our Surge program, which is inspired by the ZebraNet application [81]. Every node in ZebraNet runs of a battery that is recharged using solar cells. Since normal operation draws more power than the solar cells can supply, just before the battery is empty the node goes into sleep mode and recharges the battery. In ZebraNet this behavior is implemented by regularly checking the remaining battery level. Should it drop below a certain threshold, the node initiates charging, during which normal operation is suspended.

Checking the battery status and charging can be considered two distinct program phases, as depicted in Fig. 4.6. These phases, CHECK and CHARGE, may have multiple sub-phases each. Combined into the BATTERY_OPERATION super-phase) they run concurrently with the ENVIRONMENT_MONITORING super-phase (i.e., the rest of the system with the exclusion of the IDLE phase). □

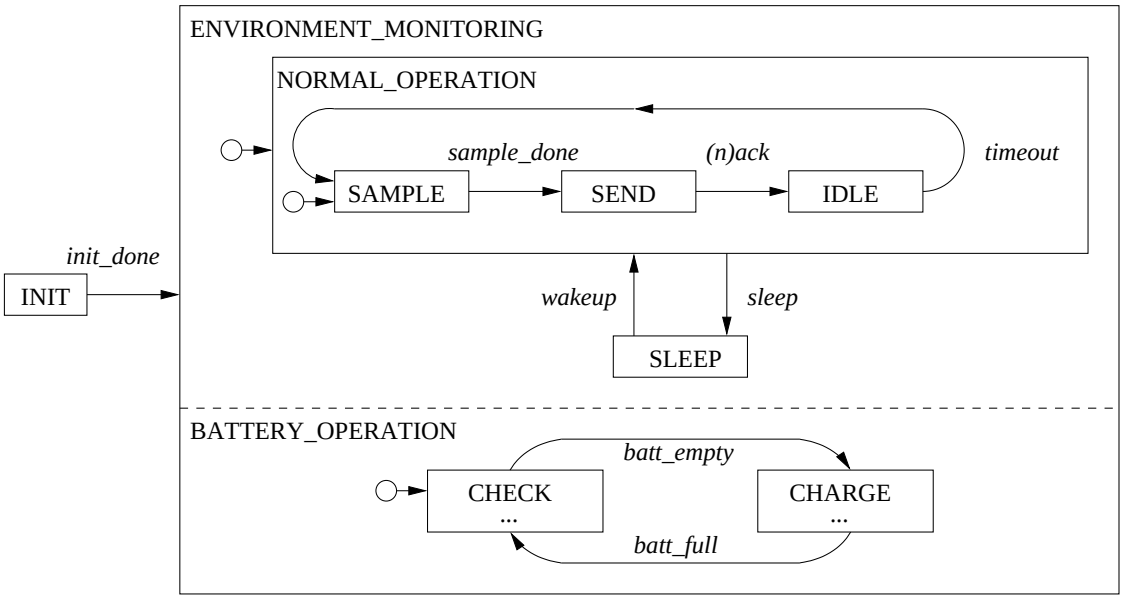



Figure 4.6: A version of Surge with two concurrent phases ENVIRONMENT_MONITORING and BATTERY_OPERATION.

### Interaction Between Phases: Local Communication

Typically the various concurrent phases of sensor-node programs do need to communicate. They run mostly independently of each other (after all, that was the reason why to model them as concurrent phases), but from time to time, they may still require some synchronization. In sensor node programs two principal methods of communication between concurrent phases are used: communication with generated events and transmitting information in data structures residing shared memory. The latter form is associated with polling while the former is associated with handling event notifications, the basic concept of event-driven programming.

***Synchronization with Events.*** In our example as depicted in Fig. 4.6, whenever the battery needs recharging the program enters the CHARGE state. To suspend normal operation when recharging, the battery subsystem needs to communicate with the concurrent monitoring subsystem. There are two possibilities of how to synchronize the two concurrent subsystems with events: Firstly, the monitoring subsystem is programmed to react to $batt\_empty$ events by preempting normal operation and by going to sleep. The second possibility is to *generate* a $sleep$ event when making the transition from checking to charging in the battery subsystem. Likewise, whenever a remote user instructs the node to enter sleep mode through the network-command interface, the node could (and should) start charging the battery. In this example, the phases ENVIRONMENT_MONITORING and BATTERY_OPERATION are synchronized directly through events, their sub-phases (NORMAL_OPERATION and SLEEP as well as CHECK and CHARGE) progress in lock step.

***Synchronization with Shared Memory and Events.*** Concurrent phases can communicate using shared memory. To avoid polling, shared-memory communication is often initiated by an event notification. Only after receiving a notification the program reads a shared memory location. In our BTnode system software, for example, the Bluetooth driver can be considered as running concurrently to the user application. The user application relies on notifications by the Bluetooth driver to indicate network-state changes, such as remote connection establishment or link failure. The driver does so by generating an event specific to the state change. In the user program the generated event then triggers a reaction. The user program typically requires more information about the event (such as the identity of the connecting node or the identity of the failed link destination). This information can be be accessed through shared memory, concretely, through a shared *connection table*, a data structure which is maintained by the Bluetooth driver but can be accessed by the application as well.

### The Absence of Dynamic Task Creation

In our study of existing sensor-node programs we have found several examples where program phases run concurrently. But we have not found programs that require the dynamic creation of (a previously unknown number of) tasks. Multithreading environments are often motivated with a Web-server example, where individual HTTP-requests are serviced by a separate and possible dynamically

created threads. (cf. [108, 115]). In such situations, the number of active tasks at any point during runtime is not known at compile time.

In sensor networks, however, programs are modeled with a clear conception how of many concurrent subsystems (i.e., program phases) there are and when they are active during the runtime of the program. The only circumstances where we did find a form of dynamic creation of concurrent program phases where errors in the program specification, such as those described as accidental concurrency in Sect. 4.1.

### 4.2.4 Sequential vs. Phase-Based Programming

Most algorithms are described as hierarchical and concurrent phases in the WSN literature. However, since there are no adequate programming abstractions and language elements to specify phases in actual programs, it is typically very hard to recognize even a known phase structure of a given program.

In many respects the phases of sensor-node programs are comparable to functions of a sequential, single threaded program. Sequential programs are composed of function hierarchies, whereas sensor node programs are composed of phase hierarchies. Functions may have private data structures (typically specified as local variables) but may also operate on shared data (the local variables of a common higher function hierarchy or global variables). Program phases of sensor-node program also have these private and shared data structures, however, programming abstractions to specify them are missing.

The main difference of functions and phases is that the duration of a function is typically determined by the speed of the CPU, whereas the duration of a phase mostly depends on the occurrence of events, which may happen unpredictably. Preemption in sensor-node programs can best be compared to exception handling of object-oriented languages. In object-oriented programming, an exception may unroll a hierarchy of nested methods calls to trigger an exception handler on a higher hierarchy level. Likewise, in phase-based sensor-node programs, an event may preempt a hierarchy of nested phases (i.e., sub-phases) and trigger a phase on a higher hierarchy level.

## 4.3 Extending the Event Model: a State-Based Approach

In this chapter we have laid down why the event-driven model is inadequate to model and specify real-world sensor-node applications. The fundamental reasons for this inability are manual state and manual stack management, that is, the need for programmers to meticulously specify the program's control flow and memory management. However, to be suitable for real-world application programming, a sensor-node programming model must be expressive enough to easily and concisely specify the anatomy of sensor-node programs, requiring neither manual state management, nor manual stack management.

### 4.3.1 Automatic State Management

First off all, to avoid manual state management, a sensor-node programming model requires an explicit abstraction of what we have presented as phases: a common context in which multiple actions can perform. Furthermore it requires mechanisms to compose phases sequentially, hierarchically, and concurrently.

A natural abstraction of program phases are *states* of finite state machines (FSM). Finite state machines—in their various representations—have been extensively used for modeling and programming reactive embedded systems. Reactive embedded systems share many of the characteristics and requirements of sensor nodes, such as resource efficiency and complex program anatomies.

Particularly the Statecharts model and its many descendants have seen widespread use. The Statecharts model combines elements form the event-driven programming model (events and associated actions) as well as from finite state machines (states and transitions among states). To these elements it adds state hierarchy and concurrency. As such, the Statecharts model is well suited to describe sensor-node programs. Statecharts can help to solve the manual state-management issues. Particularly, it allows to specify the program's control flow in terms of the model's abstractions (namely transitions between possibly nested and concurrent states), rather then requiring the programmer to specify the control flow programmatically. Indeed, we have already used the graphical Statechart notation to illustrate the flow of control through phases in the examples throughout this chapter. The Statechart model builds the foundation for our sensor-node programming model, which we will present in the next chapter.

### 4.3.2 Automatic Stack Management

However, neither the Statecharts model nor any of its descendants do solve the issue of manual stack management. As we have detailed previously, phases of sensor-node programs have associated data state. Statecharts and Statechart-like models typically do not provide an abstraction for such data state. The few models which do, however, lack a mechanism to manage such state automatically, that is, to initialize the data state upon state entry (e.g., by allocating and initializing a state specific variable) and to release it upon exit. The same holds for all (de)initializations related to phases, such as setting up or shutting down timers, switching on or off radios and sensors, etc. In general, Statechart and descendant programming models do not support the (de)initialization of states.

To fix these issues, we propose to make two important additions to the basic Statechart model: Firstly, states have dedicated initialization and deinitialization functions. These functions are modeled as regular actions. They are associated with both a state and a transition, and are executed within the context of the associated state. Secondly, states can have associated data state. This state can be considered as the local variables of states. The lifetime and scope of *state variables* are bound to their state. State variables are an application of general (de)initialization functions: data state is allocated and initialized in the state's initialization function and released upon the state's exit in its deinitialization function. (In our proposed model however, initialization and release of state variables is transparent to the programmer.) However, state variables are more

than semantic sugar: modeling data state with the explicit state-variable abstraction allows compile-time analysis of the program's memory requirements and only requires static memory management for their implementation in the system software—two invaluable features for sensor-node programing.

# 5 The Object-State Model

In the previous chapter we have described the issues of the event-driven programming model, namely manual state management and manual stack management. These issues often lead to memory inefficient, unstructured, and unmodular programs. Also, we have argued that the way programmers think about their applications (what we might call the mind model of an application) does not translate very well into the event-driven programming model. Concretely, we have diagnosed the lack of a programming abstraction for program phases, which allows to structure the program's behavior and data state along the time domain.

In this chapter we will present the OSM state-based model, which we have designed in order to attack these problems. OSM allows to specify sensor-node programs as state machines, where program phases are modeled as machine states. OSM is not a complete departure from the event-driven programming model. Rather, OSM shares many features of the event-driven model. Just as in the event-driven model, in the OSM model progress is made only on the occurrences of events. Likewise, computational actions in OSM are specified as actions in a sequential programming language, such as C. And just as in the event-driven model, OSM actions run to completion.

We like to see OSM as an extension of the event-driven model with state-machine concepts. The first of the two main advancements of OSM over the event-driven model is that it allows to specify the control flow of an application on a higher abstraction level, saving the programmer from manual state management. Through the explicit notion of states, the association of events to actions is no longer static. Rather, the program's current machine state defines the context in which an action is executed. As a result, OSM programs can be specified in a more modular and well-structured manner. The second advancement of OSM are state variables, which save programmers from manually managing temporary data state (i.e., manual stack management). State variables allow to attach data state to explicitly-modeled program states. As the scope and lifetime of state variables is limited to a state (and its substates), the OSM runtime environment is able to automatically reclaim the variable's memory upon leaving the state. As a result, OSM programs are generally more memory efficient than conventional event-driven programs. In the following subsections we will introduce OSM and informally define its execution model. We have already presented a preliminary version of OSM in [74]. The version presented here has undergone some refinements and minor extensions. In Sect. 5.1 we will present well-known state machine concepts on which OSM is built. In Sect. 5.2 we present the basic form of OSM state machines. We explain how the real-time requirements of sensor-node programs map to the discrete time model of state machines in Sect. 5.3. In Sect. 5.4 and Sect. 5.5, respectively, we will present hierarchy and concurrency as a more advanced means to structure OSM state

machines. State variables are explained in Sect. 5.6. Finally, in Sect. 5.7 we summarize the chapter.

## 5.1 Basic Statechart Concepts

The semantics and representation (i.e., language) of OSM are based on finite state machines (FSMs) as well as concepts introduced by Statecharts [53] and its descendants. Hence, before discussing OSM in the next sections of this chapter, we briefly review these concepts this section.

Finite state machines are based on the concepts of *states*, *events*, and *transitions*. A FSM consists of a set of states and a set of transitions, each of which is a directed edge between two states, originating form the source state and directed towards the target state. A machine can only be in one state at a time. Transitions specify how the machine can proceed form one state to another. Each transition has an associated event. The transition is taken (it "fires") when the machine is in the transition's source state and its associated event occurs. FSMs can be thought of as directed, possibly cyclic graphs, with nodes denoting states and edges denoting transitions.

Statecharts enhance basic FSMs with abstractions in order to model and specify reactive computer programs. As in event-based programming, Statecharts introduce *actions* to finite state machines in order to specify computational (re)actions. Conceptually, actions are associated either with transitions or with states. For the definition (i.e., implementation) of actions, most programming frameworks based on Statechart models rely on a host language, such as C. Even though a program could be fully specified with those four concepts explained above, [53] argues that the representation of complex programs specified in this naive fashion suffers from state explosion. To alleviate this problem, Statecharts introduce *hierarchy* and *concurrency* to finite state machines. A state can subsume an entire state machine, whose states are called *substates* of the composing state. The composing state is called *superstate*. This mechanism can be applied recursively: superstates may themselves be substates of a higher-level superstate. The superstate can be seen as an abstraction of the contained state machine (bottom-up view). The state machine contained in the superstate can also be seen as a refinement of the superstate (top-down view). Though a superstate may contain an entire hierarchy of state machines, it can be used like a regular (i.e., uncomposed) state in any state machine. Finally, two or more state machines can be run in parallel, yet communicate through events. State machines that contain neither hierarchy nor concurrency are called *flat*.

A concept which has received little attention so far are *state variables*, which hold data that is *local to a state or state hierarchy*. Most state-based models do not provide an abstraction for data state but instead rely entirely on variables of their programming framework's host language. In the few other models that encompass variables, the variable scope and lifetime is global to an entire state machine.

OSM is based on the conceptual elements presented in the previous section, namely states, events, actions, and state variables. However, OSM differs significantly from Statecharts and its derivates in the semantic of state variables as

well as actions. These semantics are fundamental for mitigating the problems of the event-driven model for sensor-node programming. Therefore, applying existing state-based models to the domain of sensor networks would not be sufficient.

## 5.2 Flat OSM State Machines

Fig. 5.1 illustrates a flat state machine in OSM. In practice, OSM state machines are specified using a textual language, which we will present in Chap. 6. Here we will use a graphical notation based on the graphical Statechart notation for clarity.

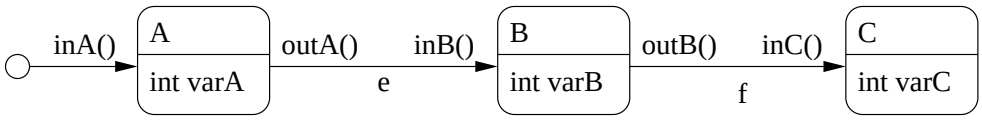



Figure 5.1: A flat state machine.

The sample OSM state machine depicted in Fig. 5.1 consists of three states $A$, $B$, and $C$. In the graphical notation, states are denoted by rounded rectangles. The initial state—the state in which the execution of the state machine begins—is denoted by an arrow sourced in a small circle. In the example, $A$ is the initial state.

### 5.2.1 State Variables

To each of these states, an integer variable $varA$, $varB$, and $varC$ is attached, respectively. These so-called *state variables* are a distinct feature in OSM. The scope of state variables is confined to their associated state (and all its substates, if any, see Sect. 5.6). OSM supports both primitive data types (e.g., integer types, characters) as well as structured data types (e.g., arrays, strings, records) in the style of existing typed programming languages, such as C.

### 5.2.2 Transitions

In the example, transitions exist between states $A$ and $B$ (triggered by the occurrence of event $e$) and between states $B$ and $C$ (triggered by the occurrence of event $f$). In OSM, events may also carry additional *parameters*, for example, sensor values (not depicted here). Each event parameter is assigned a type and a name by the programmer. Actions can refer to the value of an event parameter by its name. Typically, a transition is triggered by the event that the transition is labeled with. Additionally OSM allows to *guard* transitions by a predicate over event parameters as well as over the variables in the scope of the source state. The transition then only fires if the predicate holds (i.e., evaluates to true) on the occurrence of the trigger event.

### 5.2.3 Actions

In the sample state machine each transition between two states has two associated actions. The transition between $A$ and $B$, for example, is tagged with two actions $outA()$ and $inB()$. In OSM, each action is not only associated with a transition or a state (as in regular Statechart models) but with both, a transition and with either the source or the target state of that transition. Actions can access the variables in the scope of their associated state. Actions associated with a transition's source state are called *exit actions* (such as $outA()$ in the example) while actions associated with the target state are called *entry actions*. Entry actions can be used to initialize the data state (i.e., state variables) and other (hardware) resources of the program phase that is modeled by the target state. Entry, exit, or both actions, as well as their parameters could also be omitted.

Let us now consider the transition between $A$ and $B$ in more detail, which is tagged with two actions $outA()$ and $inB()$. When the transition fires, first $outA()$ is executed and can access $e$ and variables in the scope of state $A$. Then, after $outA()$ has completed, $inB()$ is executed and can access $e$ and variables in the scope of state $B$ (i.e., $varB$). If source and target states of a transition share a common superstate, its variables can be accessed in both actions.

Initial states can also declare entry actions, such as $inA()$ of state A in the example. Such actions are executed as soon as the machine starts. Since there is no event associated with the entrance of the initial state, $inA()$ can only access the state variable of $A$, that is, $varA$.

## 5.3 Progress of Time: Machine Steps

Finite state machines imply a discrete time model, where time progresses from $t$ to $t + 1$ when a state machine makes a transition from one state to another. Consider Fig. 5.2 for an example execution of the state machine from Fig. 5.1. At time $t = 0$, the state machine is in state $A$ and the variable $varA$ exists. Then, the occurrence of event $e$ triggers a state transition. The action $outA()$ is still performed at $t = 0$, then time progresses to $t = 1$. The state machine has then moved to state $B$, variable $varB$ exists and the action $inB()$ is executed. When event $f$ occurs, $outB()$ is performed in state $B$, before time progresses to $t = 2$. The state machine has then moved to state $C$, variable $varC$ exists and the action $inC()$ is executed. The discrete time of OSM state machines is associated with the occupancy of states.

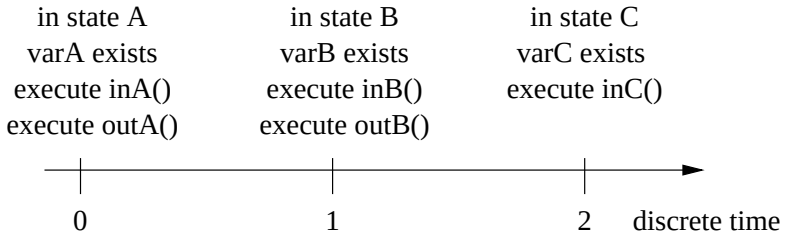



Figure 5.2: Discrete time.

Approaching time from the perspective of a sensor network that is embedded into the physical world, a real-time model seems more appropriate. For

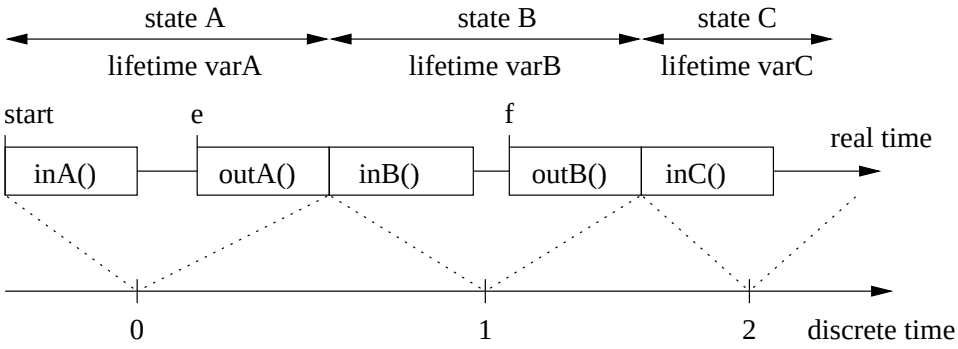



Figure 5.3: Mapping real-time to discrete time.

example, sensor events can occur at virtually any point in time, and actions and transitions require a finite amount of real-time to execute. Hence, it becomes also important in which order actions are executed. Essentially, we are faced with the issue of interfacing the discrete time model of state machines with the real-time model of sensor networks. We will discuss this interface in this and the subsequent sections.

Inspired by traditional event-based programming models, actions are considered atomic entities that run to completion without being interrupted by the execution of any other action. Fig. 5.3 illustrates the mapping of real-time to discrete time, the respective states and variable lifetimes. Since the execution of actions consumes a non-zero amount of real time, events may arrive in the system while some action is currently being executed. Those events cannot be handled immediately and must be stored temporarily. Therefore, arriving events are *always* inserted into a FIFO queue using the queue's *enqueue* operation. Whenever the system is ready to make the next transitions, that is, when all exit and entry actions have been completed, the system retrieves the next event from the queue (using the *dequeue* operation) in order to determine the next transition.

While state transitions are instantaneous in the discrete time model, they consume a non-zero amount of real time. The entire process of making a state transition is called a state-machine *step*. A step involves dequeuing the next event, determining the resulting transition(s), executing the exit action(s) of the current state(s), and executing the entry action(s) of the subsequent state(s). Also, in actions, events may be emitted programmatically, which are inserted into the event queue. A step always results in the progression of discrete time. Except for self-transitions, steps always lead to state changes. If the event queue still holds events after completing a step, the next step starts immediately. If, however, the event queue is empty after competing a step, the system enters sleep mode until the occurrence of the next event.

### 5.3.1 Non-determinism

OSM state machines may exhibit non-deterministic behavior, a characteristic is shares with other event-based systems. The reason for this non-determinism is that the exact sequence of steps performed not only depends on the reception order of input events but also on their timings and on the duration of performing a step. The amount of real-time consumed by a step depends on three factors:

the implementation of the underlying system software (for event handling and determining transitions), the actions involved in the step (which may require more or less computation), and the speed of the executing processor.

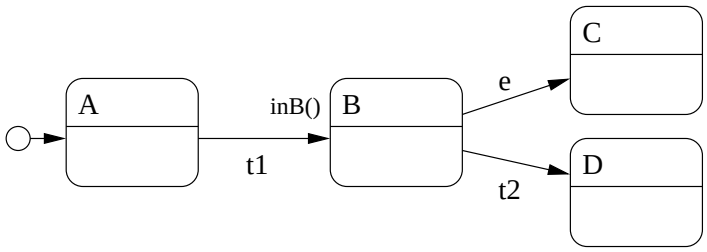



Figure 5.4: State machines that programmatically emit events may exhibit non-deterministic behavior. The event $e$, which is emitted by $inB()$, may be inserted before or after an event $t2$ depending on the execution speed of the action.

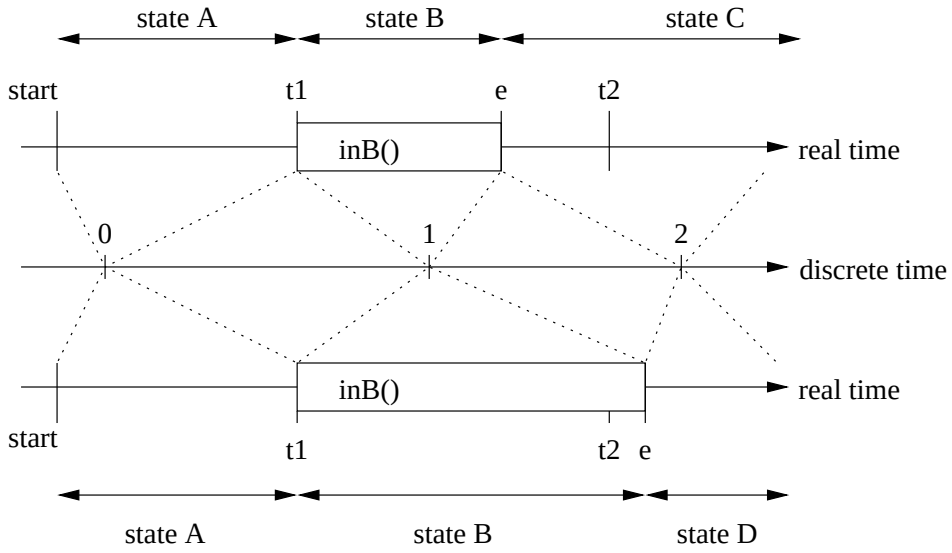



Figure 5.5: Non-determinism in OSM: the order of state transitions (and thus the invocation order of actions) may depend on the execution speed of (previous) actions. The top part of the figure depicts a possible run of the OSM state machine in Fig. 5.4 on a fast processor, the bottom part on a fast processor.

Consider, for example, two runs of the state machine depicted in Fig. 5.4. The machine has a single computational action $inB()$, which programmatically emits an event $e$. Let us now consider two runs of that state machine on sensor nodes having processors with different speeds. In both runs two input events ($t1$ and $t2$) are triggered externally, exactly $t1$ and $t2$ time units from the start of the machine. The top part of Fig. 5.5 depicts a possible result on the faster processor, while the bottom part of the figure depicts a possible result on the slower processor. On the faster processor, $inB()$ terminates earlier and $e$ is inserted into the queue before the occurrence of $t2$, whereas on the slower processor $inB()$ only terminates after the occurrence of $t2$ and thus $t2$ is inserted first. Thus, both runs end up in different states. State $C$ is reached on the fast processor and state $D$ is reached on the slow processor.

Note that OSM's nondeterminism does not stem from its transitions semantics, which are unambiguous. Rather, nondeterminism is introduced by race conditions when programmatically emitting events. Such events may be inserted into the event queue before or after a externally-triggered event, depending on the processor's execution speed. Also note that event-driven programs exhibit non-deterministic behavior for the same reason.

A possible solution to this kind of non-determinism would be to always insert programmatically-emitted events at the beginning rather than the end of the event queue. This way there would be no race conditions between internally and externally emitted events. However, this behavior is in conflict not only with the event-driven model, where events are often use to trigger some low-priority action at a later time. It may also be in conflict with the synchronous execution model. In the synchronous model, where the output from the system is always synchronous to its input, the programmatically-emitted event as well as the original input event (events $e$ and $t1$ in the example, respectively) would have to be concurrent. This could to a situation where the original transition would have not been triggered, namely if the source state had a higher-priority transition triggered by the programmatically emitted event.

### 5.3.2 Processing State Changes

A principal problem of all programs dealing with external inputs is that the internal program state may not represent the state of the environment correctly at all times. That is because real-world events require some time until they are propagated through the system and are reflected in the system's internal state. This is particularly true when events are propagated through network messages or are being generated from sensor input through multiple stages of processing. This is a principle problem and applies to the event-driven, the state-based, and the multi-threaded programming model alike.

Consider the case where the program is writing to an already closed network connection because the $disconnect$ event has not been processed (for example, the network packet containing the disconnect request has not completely received or the $disconnect$ event is lingering in the queue). In this case the program would still consider the connection to be open and may try to send the data over already closed connection.

The actual problem is that programmers must handle those cases, which typically result in errors. As these cases typically occur only rarely, they are hard to test systematically and may be forgotten entirely during program design. From our experience, error handling of these cases can become very complex. However, if they occur and error handling fails, the node may crash or enter some undefined state.

### 5.3.3 No Real-Time Semantics

The OSM model is not a real-time model—it does not allow to specify deadlines or analyze execution times. It neither has means to quantify execution times nor can they be specified by the programmer of OSM state machines.

## 5.4 Parallel Composition

An important element of OSM is the support for parallel state machines. In a parallel composition of multiple flat state machines, multiple states are active at the same time, exactly one for every parallel machine. Let us consider an example of two parallel state machines, as depicted in Fig. 5.6. (This parallel machine is the flat state machine from Fig. 5.1 composed in parallel with a copy of the same machine where state and variable names are primed.) At every discrete time, this state machine can assume one out of 9 possible state constellations with two active states each—one for each parallel machine. These constellations are $A|A'$, $A|B'$, $A|C'$, $B|A'$, $B|B$ and so forth, where $A|A'$ is the initial state constellation. For simplicity we sometimes say $A|A'$ is the initial state.

Parallel state machines can handle events originating from independent sources. For example, independently tracked targets could be handled by parallel state machines. While events are typically triggered by real-world phenomena (e.g., a tracked object appears or disappears), events may also be emitted by the actions of a state machine to support loosely-coupled cooperation of parallel state machines.

Besides this loose coupling, parallel state machines can be synchronized in the sense that state transitions of concurrent machines can occur concurrently in the discrete time model. In the real-time model, however, the state transitions and associated actions are performed sequentially.

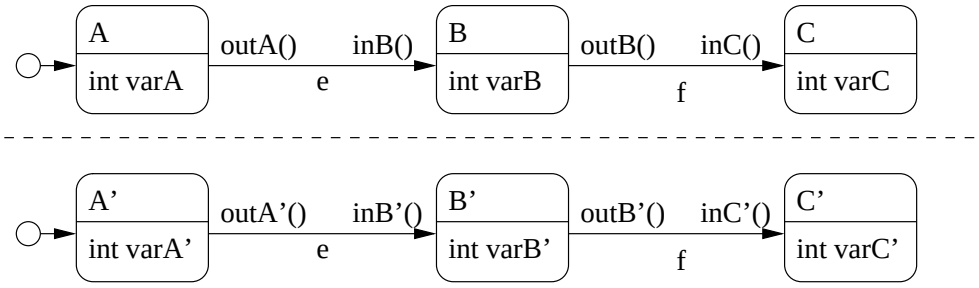



Figure 5.6: A parallel composition of two flat state machines.

The parallel state machines of Fig. 5.6 are synchronized through events $e$ and $f$, which trigger transitions in both concurrent machines. Fig. 5.7 shows the mapping of real-time to discrete time at discrete time $t = 1$. After discrete time has progressed to $t = 1$ on the occurrence of $e$, all entry actions are executed in any order. When event $f$ arrives, all exit actions are performed in any order.

### 5.4.1 Concurrent Events

Up to now, each event has been considered independently. However, in the discrete time model, events can occur concurrently when concurrent machines programmatically emit events in a synchronized step. For example, when parallel state machines emit events in exit actions when simultaneously progressing from time $t = 1$ to $t = 2$ (see Fig. 5.7), the emitted events have no canonical order. The event queue should preserve such unordered sets of concurrent events, rather than imposing an artificial total ordering on concurrent events by inserting them one after another into the queue. For this purpose, the event queue

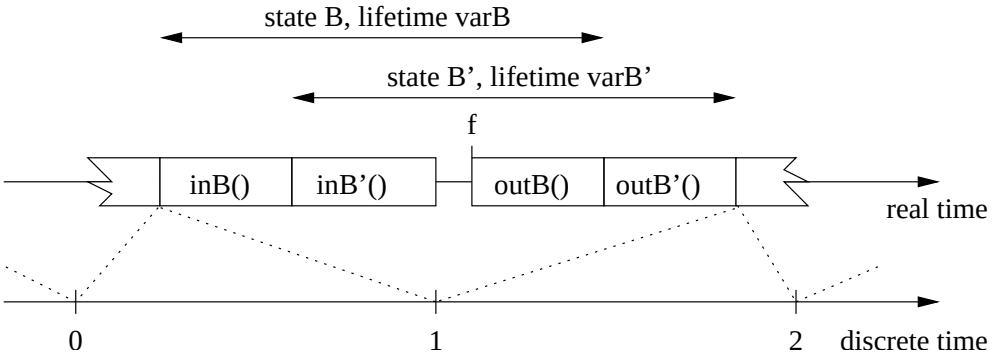



Figure 5.7: Possible mapping of real-time to discrete time for parallel machines.

of OSM operates on *sets of concurrent events* rather than on individual events. All elements of a set are handled in the same discrete state-machine step. The enqueue operation takes a set of concurrent events as a parameter and appends this set as a whole to the end of the queue. The dequeue operation removes the set of concurrent events from the queue that has been inserted earliest. Whenever the system is ready to make a transition, it uses the dequeue operation to determine the one or more events to trigger transitions.

All events emitted programmatically in exit and entry actions, respectively, are enqueued as separate sets of events, even though they are emitted during the same state machine step. That is, each state-machine step consumes one set of events, and may generate up to two new sets: one for events emitted in exit actions and another set for events emitted in entry actions. Clearly, the arrival rate of event sets must not be larger than the actual service rate, which is the responsibility of the programmer.

### 5.4.2 Progress in Parallel Machines

The execution semantics of a set of parallel state machines can now be described as follows. At the beginning of each state-machine step, the earliest set of concurrent events is dequeued unless the queue is empty. Each of the parallel state machines considers the set of dequeued (concurrent) events separately to derive a set of possible transitions. Individual events may also trigger transitions in multiple state machines. Similarly, if multiple concurrent events are available, multiple transitions could fire in a single state machine. To resolve such ambiguous cases, priorities must be assigned to transitions. Only the transition with the highest priority is triggered in each state machine. Actions are executed as described in Sect. 5.3. The dequeued set of concurrent events is then dropped.

## 5.5 Hierarchical Composition

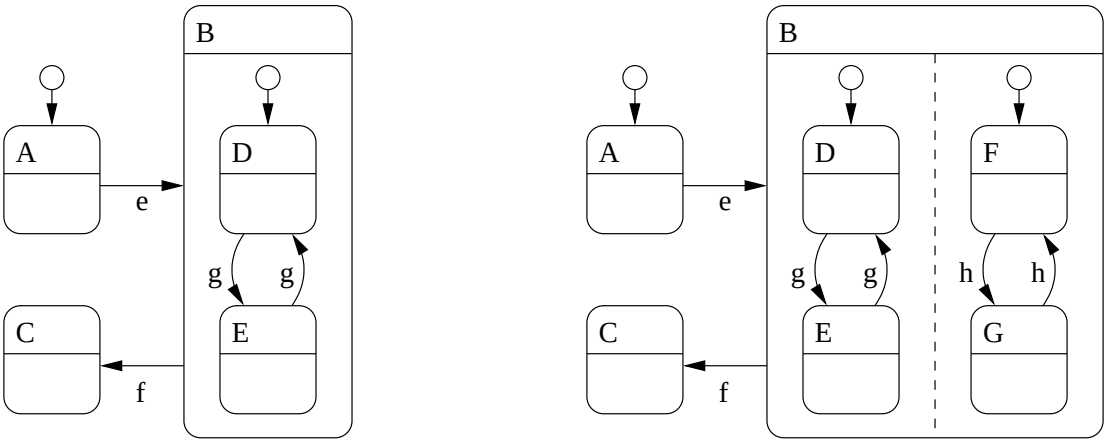
Another important abstraction supported by OSM are state hierarchies, where a single state (i.e., a superstate) is further refined by embedding another state, state machine (as depicted in Fig. 5.8 (a)) or multiple, concurrent state machines (as depicted in Fig. 5.8 (b)). Just as uncomposed states, superstates can have state variables, can be source or target of a transition, and can have entry and

exit actions defined. Note, however, that unlike Statecharts, in OSM transitions may not cross hierarchy levels. (Although transitions across hierarchy levels may give the programmer more freedom in the specification of actual programs, we believe that this feature impairs modularity.)

The lifetime of embedded state machines starts in their initial state when their superstate is entered. The lifetime of embedded state machines ends when their superstate is left through a transition.

The lifetime of state variables attached to a superstate is bound to the lifetime of that superstate. The scope of those variables recursively extends to all substates. In other words, an action defined in a state can access the variables defined in that state plus the variables defined in all of its superstates. State variables in hierarchically-composed state machines are further discussed in Sect. 5.6.

As in parallel state machines, in hierarchical state-machines multiple states are active at the same time. The constellation of active states contains the hierarchy's superstate plus a single state or state constellation (in the case of parallel machines) on every hierarchy level. The active constellation of a state machine can be represented as a tree, either graphically or textually. For example, starting from the initial state $A$, the state machine depicted in Fig. 5.8 (a) assumes the state constellation $B(D)$ after receiving event $e$. In $B(D)$, both states $B$ and $D$ are active and $D$ is the substate of $B$. The state machine depicted in Fig. 5.8 (b) assumes the state constellation $B(D|F)$ in the same situation.




(a) A flat state machine embedded into state $B$

(b) A parallel state machine embedded into state $B$

Figure 5.8: Hierarchies of state machines

### 5.5.1 Superstate Entry

When a superstate is entered (either through a transition or at the start of the machine), also the initial state of the embedded state machine is entered in the same atomic step. If the initial state of the embedded state machine is refined further, this applies recursively. Fig. 5.9 illustrates the mapping of real-time to discrete time of the state machine depicted in Fig. 5.8 (b) while progressing from

the initial state $A$ to the state constellation $B(D|F)$. For this example, assume that each transition has both an exit and entry action named $outS()$ and $inT()$, respectively, where $S$ is a placeholder for the transition's source state and $T$ is a placeholder for the transition's target state. On the occurrence of event $e$, the exit action of state $A$ is executed, still at discrete time $t = 0$. Then the discrete time advances to $t = 1$ and the state constellation $B(D|F)$ is entered. With becoming active, the entry action of each state of the active constellation is executed. Entry actions of states higher in the hierarchy are invoked before those of states lower in the hierarchy. Entry actions of states on the same hierarchy level (that is, entry actions of states of parallel machines) are executed in any order, as described in Sect. 5.4.2.

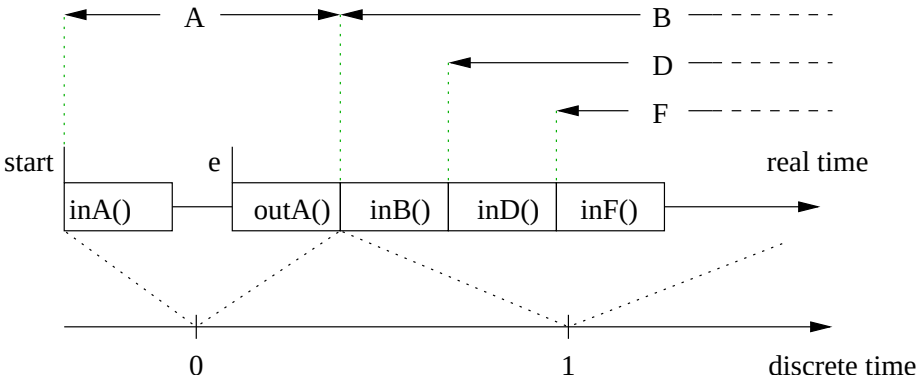



Figure 5.9: Mapping of real-time to discrete time when entering the constellation $B(D|F)$ of the state machine depicted in Fig. 5.8 (b) from state $A$.

### 5.5.2 Initial State Selection

Up to now we have only considered state machines with a single initial state on each hierarchy level. However, OSM state machines may have multiple potential initial states per level, of which exactly one must be selected upon superstate entry. Which of the potential initial state is actually selected depends on a condition that can be specified by the programmer. In OSM, these conditions are modeled as guards. In the graphical version of OSM, these guards have an intuitive representation: they are placed on the transitions from the small circle to the initial states. Fig. 5.10 illustrates the conditional entry to one of the substates $D$ and $E$ of superstate $B$. Which of $B$'s substates is actually entered depends on the value of $a$. If $a = 1$, substate $D$ is entered, $E$ otherwise. Note that it is not possible to prevent or delay substate entry. If a superstate is entered, exactly one of its substates must be entered. Therefore the guard conditions must be discrete.

As regular guards are initial state conditions, they may be composed of expressions over state variables in scope, (side-effect free) computational functions, or both. However, they may not refer to any information regarding the transition that lead to the entry of the superstate (such as source state, event type, and event parameter). This may seem awkward. However, conditional state entry based on such information can be easily implemented by setting state variables in entry actions. When $B$ is entered, the state variable $a$ can be set in

the entry actions of $B$ and then be evaluated in the guard, selecting the actual initial state. This is possible since the entry actions of a superstate are always executed before the initial-state condition is evaluated.

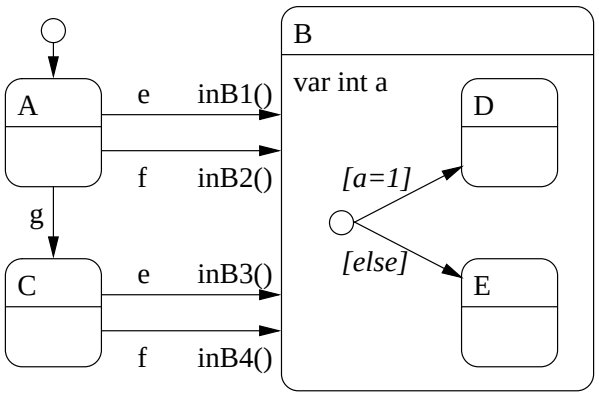



Figure 5.10: Conditional entrance to the substates $D$ and $E$ of $B$, depending on guards over the state variable $a$. The state variable $a$ can be set in the entry actions of $B$, for example, depending on which transition was taken.

### 5.5.3 Substate Preemption

When a transition fires, it ends the lifetime of its source state. We say the transition terminates the source state normally. If the source state of a transition is a superstate, the transition also preempts its substates (i.e., the embedded state machine). Then we say the transition *preempts* the source state's substates. States can only be left either through normal termination or through preemption. Substates are preempted level-by-level, starting from the bottom-most state in the hierarchy. A substate cannot prevent its superstate from being left (and thus itself from being preempted).

If a state is preempted, its exit actions do not fire as during normal termination. In order to allow the preempted state to react to the termination or preemption of its superstate (and thus its own preemption), *preemption actions* are introduced. Preemption actions are useful, for example, to release resources that where initialized in the preempted states' entry actions. Each state can have a single preemption action. This preemption action is invoked as the state is preempted. Preemption actions are invoked in the order in which their states are preempted, that is, from the bottom of the hierarchy to its top. The substates' preemption actions are invoked before the exit action of their superstate. Preemption actions of parallel states are executed in any order.

The scope of the preemption action is the to-be-preempted substate. That is, a preemption action can access all state variables its state before it is actually being preempted plus all state variables of (direct and indirect) superstates. Note however, that a transition terminating a state directly does not trigger that state's preemption action—any required operations can be performed in the regular exit action. In a preemption action, there is no means to determine any information of the actual transition causing the preemption, such as its trigger and its source state.

Let us again consider the state machine depicted in Fig. 5.8 (b), assuming every transition has both an exit and entry action named $outS()$ and $inT()$, respectively, where $S$ is a placeholder for the transition's source state and $T$ is a placeholder for the transition's target state. Let us also consider that each state $P$ has a preemption actions $preP()$. Fig. 5.11 illustrates the preemption of the constellation $B(D|G)$ through the transition from $B$ to $C$ triggered by $f$.

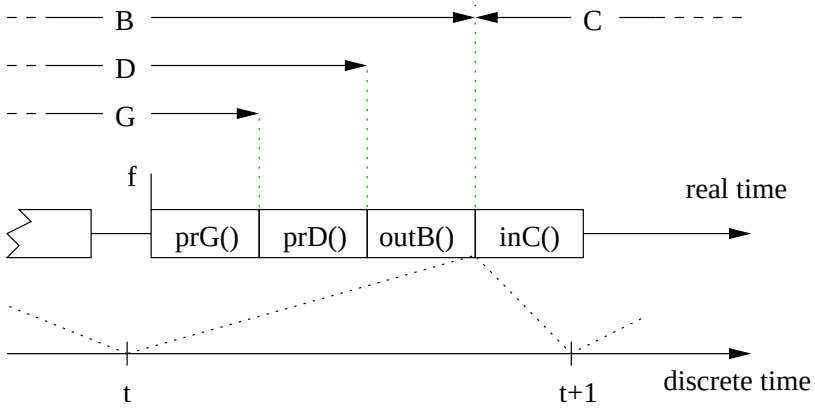



Figure 5.11: Mapping of real-time to discrete time when leaving the constellation $B(D|G)$ of the state machine depicted in Fig. 5.8 (b) to state $C$.

### 5.5.4 Progress in State Hierarchies

We can now describe the full execution semantics of a step in hierarchical and concurrent OSM state machines. Note however, that the algorithm presented here is not actually used in the implementation of OSM. It is only presented for the understanding of the reader. The actual implementation of OSM state machines is described in Sect. 6.2 of the next chapter.

At the beginning of each state-machine step, the earliest set of concurrent events is dequeued. Then the set of dequeued events is considered to derive possible transitions by performing a preorder search on the tree representing the active constellation, beginning from the root of the tree. If a matching transition is found in a state, the transition is marked for later execution and the search of the state's sub-trees (if any) are skipped. If multiple transitions could fire in a single state, only the transition with the highest priority is marked, as described in Sect. 5.4.2. Skipping subtrees after finding a match means that transitions higher in the state hierarchy take precedence over transitions lower in the hierarchy. The search continues until the entire tree has been traversed (or skipped) and all matching transitions have been marked.

Then the following steps are performed while the state machine still is in the source constellation. For each transition selected for execution, the preemption actions of states preempted by that transition are invoked in the correct order (from bottom to top). This is achieved by performing a postorder traversal on each of the subtree of the transition's source state, executing preemption actions as states are visited. The order in which entire subtrees are processed can be chosen arbitrarily, since each subtree represents a parallel machine. While still being in the source constellation, the exit actions of all marked transitions are

executed. Again, because all marked transitions are always sourced in parallel states, they have no natural order and may be executed in any order.

Next, the state machine assumes a new active constellation, that is, discrete time advances from $t$ to $t + 1$. Now, the entry actions of the target states of the marked transitions are executed. Again, they can be executed in any order. If the entered target state, however, is a superstate, the entry actions of its substates are invoked according to their level in the hierarchy from top to bottom, as previously described in Sect. 5.5.1. This is achieved by performing a preorder traversal of the subtrees sourced in transition targets, executing initial entry actions as states are visited.

Finally, the dequeued set of events is dropped. If no matching transition was found, the dequeued set of events is dropped anyhow and the machine remains in the actual constellation. Events emitted by actions during the state machine step are enqueued in analogy to Sect. 5.4.2.

Fig. 5.12 (a) depicts a state machine with transitions triggered by a single event $e$ on multiple levels. Fig. 5.12 (b) depicts the machine's initial constellation and the constellation active after performing the step triggered by event $e$. The initial constellation is shown together with the result of the in-order search for transitions triggered by $e$. States marked with either check marks or crosses are sources of matching transitions, while a missing mark denotes that there is no transition matching event $e$. Transitions actually selected for the upcoming machine step are checked. A cross denotes a state with a matching transition that is not selected (i.e., it does not fire) because a transition on a higher hierarchy level takes precedence. (In the example, this is always the case for the transition from $D$ to $E$, so the transition is redundant. However, such a situation may make sense, if one or both transitions had guards.) Before executing the transitions and their associated actions, first preemption actions are invoked. In the example, only state $D$ is preempted by the transition from $A$ to $B$. Thus $D$'s preemption action is invoked. Then the exit actions of all checked states are executed and the target constellation is assumed, as depicted in Fig. 5.12 (b). Note that all target states (i.e., states entered directly, here $B$, $G$, and $J$) are from parallel compositions. By entering state $J$, however, also its substate $K$ is entered. Entry actions of implicitly entered states are always invoked after their direct superstate.

## 5.6 State Variables

State variables hold information that is local to a state and its substates. The scope of the variables of a state $S$ extends to entry, exit, and preemption actions that are associated with $S$ and to all actions of states that are recursively embedded into $S$ via hierarchical composition. In particular, state variables are intended to store information common to all substates and share it among them.

The lifetime of state variables start when their state is entered just before invoking the state's entry actions. Their lifetime ends when the state is left (through a transition or through preemption) right after the invocation of the last exit action. With respect to variable lifetimes, there is a special case for self transitions that enter and leave the same state. Here, the variables of the affected

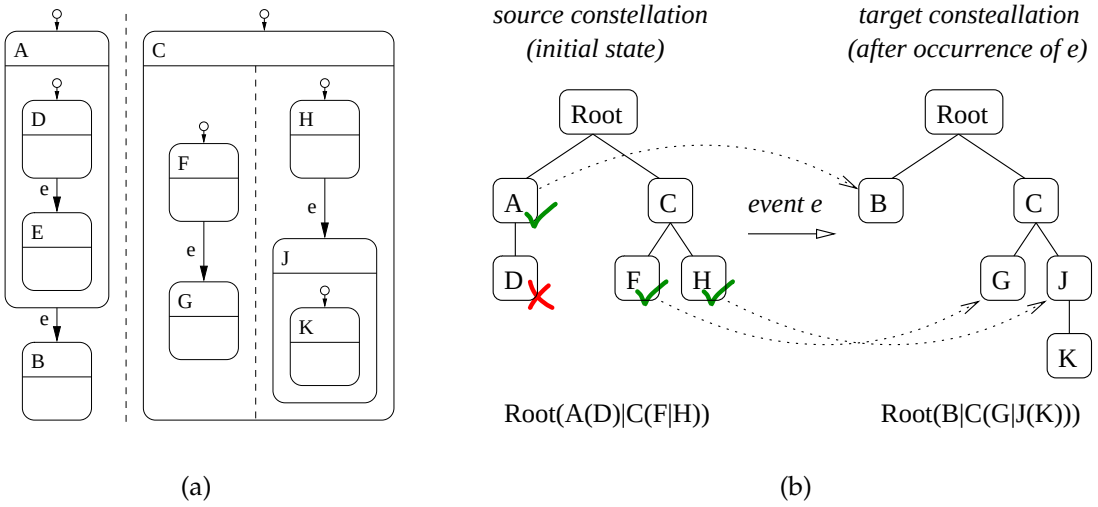



Figure 5.12: Determining the transitions to fire in a state machine step. The state machine depicted in (a) starts in its the initial constellation and $e$ is the next event from the queue. Transitions originating in $A$, $F$, and $H$ are selected for firing. The transition from $D$ is skipped as transitions higher in the hierarchy take precedence. In a non-concurrent state machine at most one transition can fire. If multiple transitions fire, they can do so in any order.

state and their values are retained during the transition, rather then deleting the variables and creating new instances.

Note that on a particular hierarchy level, a non-concurrent state machine can only assume one state at a time. Hence, only the variables of this current state are active. Variables of mutually exclusive states on the same level can then be allocated to overlapping memory regions in order to optimize memory consumption (see Sect. 6.1.1).

By embedding a set of parallel machines into a superstate, actions in different parallel state machines may access a variable of the superstate concurrently during the same state machine step at the same discrete time. This is no problem as long as there is at most one concurrent write access. If there are multiple concurrent write accesses, these accesses may have to be synchronized in some way. Due to the run-to-completion semantics of actions, a single write access will always completely execute before the next write access can occur. However, the order in which write accesses are executed (in the real-time model) is arbitrary and may lead to race conditions. Write synchronization is up to the programmer.

## 5.7 Summary

In this chapter we have informally presented OSM, a programming model for resource constrained sensor-nodes. OSM is an extension of the event-driven programming model, adding two main features to the popular programming model: firstly, an explicit abstraction of program state as well as mechanisms to

compose states hierarchically and in parallel. Therefore, in OSM, programs are specified as hierarchical and concurrent state machines. OSM allows to specify a program's control flow in terms of program states and transitions between them. The addition of these abstractions in OSM allow to specify sensor-node programs more modular and structured compared to the conventional event-driven model. The second feature builds on the first: OSM uses the explicit notion of state as scoping and lifetime qualifier for variables, as well as context for computational actions. The main achievement of state variables is the automated reuse of memory for storing temporary data by mapping data of mutually exclusive states to overlapping memory regions.

Despite these additions, OSM shares several features of the event-driven programming model. As in the event-driven programming model, progress is made only in reaction to events by the invocation of actions, which also run to completion. Furthermore, OSM also stores events in an event queue while a previous event is being processed.

# 6 Implementation

In the previous chapter we have presented the fundamental concepts and abstractions behind the Object State Model. In this chapter we will examine the four missing pieces that are needed to turn these concepts into a concrete programming framework, which allows to produce executable sensor-node programs from OSM specifications. In Sect. 6.1 we will present our programming language for OSM. This language provides the syntactic elements to make use of the concepts and abstractions found in the OSM programming model (such as state variables, state transitions, and actions) thus allowing to specify concrete programs. The OSM language is used only to specify the program's variables, structure, and control flow. Computational operations (i.e., the implementation of actions) are specified as functions of a procedural host language, such as C. These functions are called from within the OSM-language code. As such, the OSM language is not a classical programming language.

Before an executable can be created, an OSM compiler first translates the OSM-language code into the host language as well. A language mapping specifies how OSM specifications are translated into the host language. It specifies, for example, how OSM actions are translated into function signatures of the host language. The language mapping and implementation details of the compiler are presented in Sect. 6.2 and 6.3, respectively. In the final compilation step an executable is created form the host-language files by a platform-specific host-language compiler. The steps to create an executable from an OSM specification is depicted in Fig. 6.1.

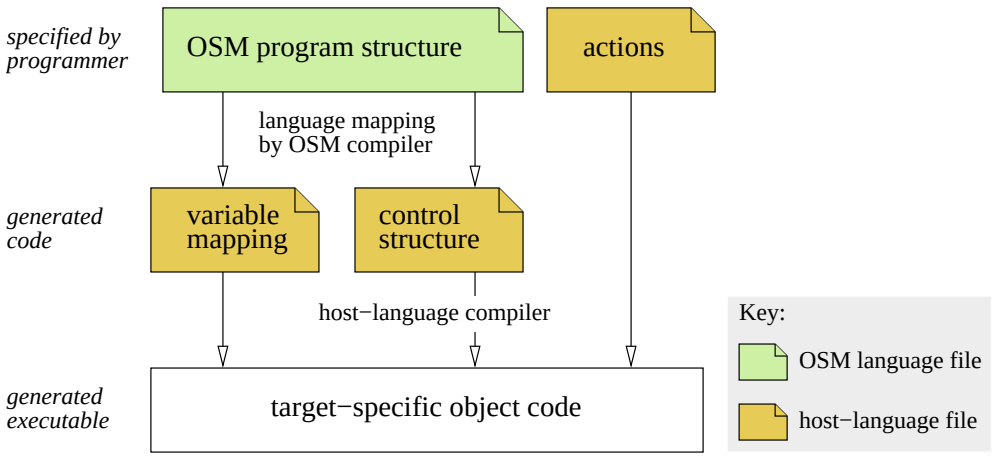



Figure 6.1: The OSM language is used to specify a program's structure and control flow as state machines, and to associate variables to states. Computational operations are specified in a host language. An OSM compiler then maps the OSM program specification to this host language as well, from which the executable is created by a target-specific host-language compiler.

When run on an actual sensor node, the generated executable then relies on support by the runtime environment. This environment must be provided by the target platform's system software. For OSM-generated executables, the only runtime support required is a basic event-driven system software capable of handling sets of events. In Sect. 6.4 we present such a system software. Its implementation is largely based on our original event-driven BTnode system software, which we have already presented in Sect. 3.5.1. Therefore we only highlight the changes to the original system.

## 6.1 OSM Specification Language

As opposed to most state machine notations, which are graphical, OSM specifications use a textual language. The following subsections present important elements of this language.

### 6.1.1 States and Flat State Machines

The prime construct of the OSM language is a state. The definition of a state includes the definition of its variables, its entry actions, its substates, its transitions (which include exit actions), and its preemption actions. Many of those elements are optional or are useful in hierarchical compositions only. We will explain the elements of state definitions as well as their use in this and the following subsections. State definitions have the following form (where $S$ is the name of the state):

**state** $S$ **{**
*state-variables*
*entry-actions*
*substates*
*transitions*
*preemption-actions*
**}**

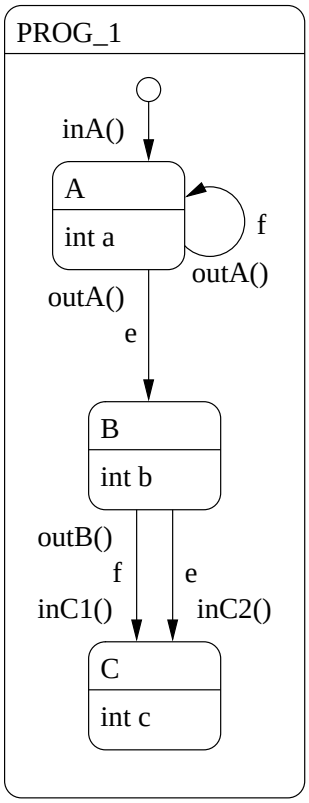
A flat state machine is composed of one or multiple states, of which at least one must be an initial state. Multiple states of a flat machine must be connected through transitions, so that every state of the machine is reachable from its initial state. In the OSM textual language, a state machine composed of multiple states must be hierarchically contained in a superstate. We will present hierarchy in Sect. 6.1.2 below. For now we will focus on flat state machines. Prog. 6.1 is an example of a simple, yet complete OSM program implementing a flat OSM state machine (lines 3-20). The state machine consists of three states, $A$, $B$, and $C$, where $A$ is the initial state (denoted by the `initial` keyword).

#### Outgoing Transitions and Exit Actions

Definitions of transitions have the following form (where $T$ is the target state, $e$ is the trigger event, $g$ is the guard, and $a_{exit}()$ is the entry action to be triggered):

**[**$g$**]**$e$ **/** $a_{exit}$**() ->** $T$ **;**

Program 6.1: A flat state machine with variables, as well as entry and exit actions in the OSM textual language.




```
1  state PROG_1 {
2
3     initial state A {
4       var int a;
5       onEntry: start/ inA();
6       e / outA() -> B;
7       f / outA() -> A;
8     } // end state A
9
10    state B {
11        var int b;
12        f / outB() -> C;
13        e /        -> C;
14    } // end state B
15
16    state C {
17        var int c;
18        onEntry: B -> f/ inC1();
19        onEntry: B -> e/ inC2();
20    } // end state C
21
22 }
```

Transitions must be defined in their source state. The definition of a transition must contain a trigger and, except for self-transitions, contains a target state. The source state of a transition does not need to be specified explicitly. The specification of a guard and an exit action is optional. Actions are defined outside of OSM in a host language. In line 6 of Prog. 6.1 a transition is defined leading from $A$ to $B$. It is triggered by event $e$. It has an exit action $outA()$ and has no guard.

State $B$ is source of two transitions declared in lines 12 and 13. Transitions have priorities according to their order of definition, with the first transition having the highest priority. For example, if two concurrent events $e$ and $f$ occur in state $B$, both transitions could fire. However, only the transition triggered by $f$ is taken, since it is declared before the transition triggered by $e$. In OSM, the selection of transitions is always deterministic.

Self-transitions are transitions with identical source and target states. In OSM there are two different ways to specify self-transitions, which have different semantics. If the target state is given in the specification of a self-transition (as in "`f / outA() -> A;`" in line 7), the transition leaves the source state and immediately re-enters it. That is, all state variables go out of scope and new instances are created. Also, the state's exit and entry actions are executed as well as preemption actions of substates. If, however, the target state is omitted, as in "`f / outA();`", the state variables and their values are retained and only the

specified action is executed. If the target state is missing, the specification of an action is mandatory.

### Incoming Transitions and Entry Actions

As stated previously, transitions are defined in the scope of the transition's source state together with the transition's exit action. In order to specify entry actions, part of the transition must be again declared in the transition's target state. That is, if a transition is to trigger both an exit and an entry action, the transition appears twice in the OSM code: It must be *defined* in the transition's source state, together with the exit action. And it must be *declared* again as incoming transition in the target state (together with the entry action). Both declarations denote the same transition, if source state, target state, trigger and guard are equal. If only the exit action of a transition is required, its declaration as incoming transition is obsolete.

The reason for this seeming redundancy is the intent to reflect OSM's scoping of actions in the programming language. In OSM, exit actions are executed in the context of a transition's source state, whereas entry actions are executed in the context of a transition's target state. This separation is reflected in the syntax and clearly fosters modularity, as changes to the incoming action do not require changes in the transition's source state (and vice versa) at the price of slightly more code.

The general form of entry actions is (where $S$ is the source state, $e$ is the trigger event, $g$ is the guard, and $a_{entry}()$ is the entry action to be triggered):

**onEntry:** $S$ **->** [$g$]$e$ **/** $a_{entry}$**();**

In Prog. 6.1, a transition from $B$ to $C$ triggered by $f$ has an entry action $inC1()$. The actual transition is defined in line 12. The corresponding entry action is declared in transition's target state $C$ in line 18. The declaration of the entry action includes the transition's source state $B$ and its trigger $f$. However, the declaration of the transition's exact source state as well as its trigger event and guard are optional and only required to identify the exact incoming transition should there be multiple incoming transitions. This information can be left out where this information is not required to identify the transition, for example if there is only one incoming transition. In our example, state $C$ has two incoming actions, both invoked on transitions from the same source state $B$. Therefore the specification of the source state is not strictly required in the declaration and line 18 could be replaced with "`onEntry: f / inC1();`" without changing the program's semantic. Missing source states, trigger events, and guards, in entry-action declarations are considered to match all possibilities. Therefore, in the declaration "`onEntry: / a();`" in the definition of a state $S$, action $a()$ would be executed each time $S$ was entered. When multiple declarations match, all matching entry actions are executed.

### Entry Actions on Initial State Entry

In order to trigger computational actions when entering the initial state of a state machine (either at the start of the entire program or because a superstate

has been entered), OSM uses incoming transitions as specified by the `onEntry` keyword. To distinguish initial state entry from the reentry of the same state (e.g., through a self transition), the pseudo event $start$ is introduced. The $start$ pseudo event is not a regular event. Particularly, it is never inserted into the event queue. For hierarchical state entry, it can be considered a placeholder for the actual event that triggered the entry of the superstate, though non of the event's parameters can be accessed. An entry action triggered by $start$ should never have a source state specified. In the example program *PROG_1*, the $start$ pseudo event is used to distinguish between the entry of state $A$ as an initial state and its reentry through the self transition specified in line 7.

#### Variable Definitions

State variables can only be defined in the scope of a state. They are visible only in that state and its substates, if any. Variables are typed and have a name, by which they can be referenced in actions anywhere in their scope. In the code of the previous example, each state defines an integer variable $a$, $b$, and $c$ (in lines 4, 11, and 17), respectively (though they are never used).

#### Actions Parameters

Actions may have numerical constants and any visible state variables as parameters. Action parameters must be declared explicitly in the declaration of an action. For example, in the code fragment below, $outA()$ has two parameters: the state variables *index* and *buffer*.

$e$ **/** $outA$**(** *index*, *buffer* **) ->** $T$ **;**

Actions map to functions of the same name in the host language. For each host language, there is a language mapping, which defines how actions and their parameters map to function signatures. Programmers must then implement those functions. An example of the mapping of an action with parameters into the C language is shown in Sect. 6.2.1 below.

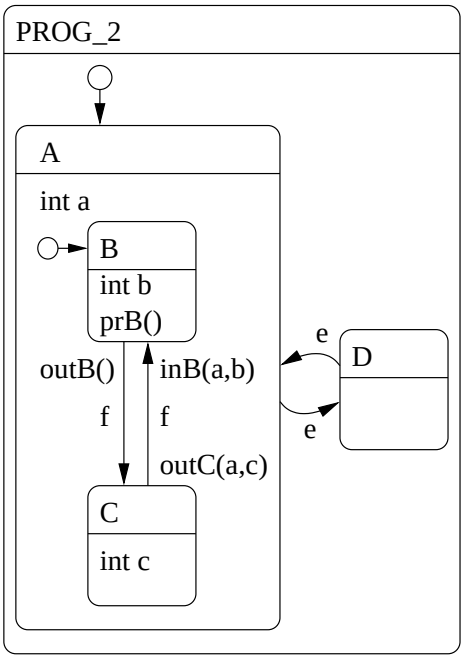
### 6.1.2 Grouping and Hierarchy

An entire state machine (composed of a single state or several states) can be embedded into a single state (which then becomes a superstate). In OSM, states can be embedded into a state simply by adding their definitions to the body of that state. States can be nested to any level, that is, substates can embed state machines themselves. In Prog. 6.2 a state machine composed of two states, $A$ and $D$, is embedded into the top-level state *PROG_2*, while $B$ and $C$ are embedded into *PROG_2*'s substate $A$ (lines 5-15).

The states of the embedded state machine must only contain transitions to states on the same hierarchy level. At least one of the states must be marked as the `initial` state. Each state of a flat state machine must be reachable through a transition from the machine's initial state.

All states of the embedded state machine may refer to variables that are defined in any of their superstates. These variables are said to be *external* of that

Program 6.2: A hierarchical state machine and its representation in the OSM textual language.




```
1  state PROG_2 {
2     initial state A {
3        var int a;
4
5        initial state B {
6           var int b;
7           onEntry: f / inB(a, b);
8           f / outB() -> C;
9           onPreemption: prB();
10       } // end state B
11       state C {
12          extern var int a;
13          var int c;
14          f / outC(a, c) -> B;
15       } // end state C
16
17       e / -> D;
18    } // end state A
19
20    state D { e / -> A; }
21 }
```

state. If desired, the external variable interface may be declared explicitly, but if it is declared, external variables must be marked with the `extern` keyword. In the example, the integer variable $a$, defined in state $A$ (line 3), is visible in $A$'s substates $B$ and $C$. Therefore $a$ can be used in their actions, such as $inB()$ and $outC()$ in lines 7 and 14 of the example Prog. 6.2, respectively. In state $C$ the external variable $a$ is declared explicitly.

### Preemption Actions

Substates may declare preemption actions that are executed at runtime when the substate is left because its superstate is left through a transition. Preemption actions are declared in the context of states—they are not explicitly associated with particular transitions. Definitions of preemption actions have the following form (where $a_{preemption}()$ is the preemption action):

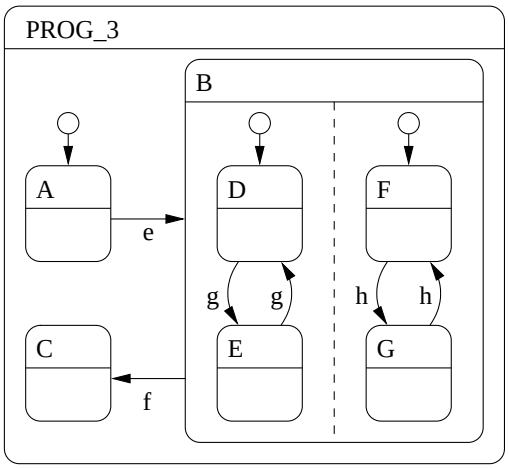
**onPreemption: /** $a_{preemption}$**();**

In the example Prog. 6.2, state $B$ declares a preemption action $prB()$ in line 9. It could be used to release resources previously allocated in the entry action $inB()$ when $B$ is left implicitly by preemption, that is, when leaving $B$'s superstate $A$ through a transition to $D$.

### 6.1.3 Parallel Composition

Parallel compositions of states can be defined by concatenating two or more states with "||". In the OSM textual language, parallel state compositions must be contained in a superstate. On the next lower level, a superstate can either contain a parallel composition of states or a sequential state machine, but never both. That means, if a superstate is to contain two parallel state machines of two or more states each, an additional hierarchy level is required. This case is shown in Prog. 6.3, which is the textual OSM representation of the state machine depicted in Fig. 5.8 (b) (the original figure is reprinted here for convenience). In the example, state $B$ contains two state machines of states $D$, $E$ and $F$, $G$, respectively. In the given OSM program, each of those machines is embedded into an anonymous state (lines 5-8 and lines 10-13; not shown in the figure), which is a direct substate of $B$. States that need not be referenced by transitions may be anonymous, that is, they need not have a name. Such states are initial states with no incoming transitions and states grouping parallel machines (as in the example).

Program 6.3: Representation of the parallel state machine depicted in Fig. 5.8 (b) (reprinted here for convenience) in the OSM textual language.




```
1  state PROG_3 {
2    initial state A { e / -> B; }
3
4    state B {
5        state {
6          initial state D { g /-> E; }
7          state E { g /-> D; }
8        }
9      ||
10       state {
11         initial state F { h /-> G; }
12         state G { h /-> F; }
13       }
14       f / -> C;
15   } // end state B
16   state C {} // end state C
17 } // end state PROG
```

### 6.1.4 Modularity and Code Reuse through Machine Incarnation

To foster modularity and the reuse of program code, OSM supports the instantiation of states. A programmer may instantiate a state that is defined elsewhere in the program (or in another file implemented by another developer) in order to substitute a state definition anywhere in the program. There are no restrictions on the state to be incarnated except that recursive instantiation is not allowed. A

state definition may be substituted using the `incarnate` keyword followed by the name of the actual state declaration. In order to adapt the actual state definition to the program context of its incarnation, a number of name-substitutions and additions can be made to the original declaration in the incarnation statement. The full declaration of a state incarnation has the following form (where $S$ is the name of the state incarnation, and $I$ is the state to be incarnated):

**state** $S$ **incarnate** $I$ (*name substitutions*) **{**
  *additional entry actions*
  *additional transitions*
  *additional preemption actions*
**}**

Additionally, state definitions may be marked with the `template` keyword in order to denote that the definition is not a complete program but instead is a template definition meant for incarnation elsewhere in the program. Though not strictly required, state templates should specify their complete interface including variables, events, and actions. The example Prog. 6.4 shows the definition of a template state $I$ (lines 1-9) and its incarnation as a direct substate of the top-level state $PROG\_4$ (lines 14-19). Additionally, $PROG\_2$ (which we have defined previously in Prog 6.2) is also inserted as a direct substate of $PROG\_4$ (lines 21-26). We will use this example to explain the various elements of the above incarnation statement in the following subsections.

### Name Substitution

A state can be incarnated as a substate of a regularly defined (i.e., non-incarnated) state. The incarnation has the same name as the incarnated state unless given a new name in the incarnation statement. If the a state definition is incarnated more than once, the incarnation should be given a new name in order to distinguish it from other incarnations. Lines 14 and 21 of Prog. 6.4 shows how to rename incarnations.

Just like regular states, incarnated states may access the variables of their superstates. The external variables of the incarnation (i.e., the ones used but not defined in the incarnation, see lines 2 and 3) are bound to the variables of the embedding superstates at compile time. External variables must have the same name and type as their actual definitions. However, since the incarnated state and the superstate may be specified by different developers, a naming convention for events and variables would be needed. In order to relieve programmers from such conventions, OSM supports name substitutions for events and external variables in the incarnation of states. This allows to integrate states that were developed independently. Line 15 of Prog. 6.4 shows how the external variable $x$ of the original definition (line 2) is renamed to $u$ in the incarnation. Renaming of events is done analogously. Lines 16, 22, and 23 show how events are renamed. Note that in the incarnation of *PROG_2* the event names $e$ and $f$ are exchanged with respect to the original definition.

### Adding Entry Actions and Transitions

When incarnating a state, transitions, entry actions, exit actions, and preemption actions can be added to the original state definition. Adding transitions and actions to the original state definition is typically used when the program context of the incarnation is not known at implementation time. In Prog. 6.4, $A$ is the incarnation of template $I$. In $I$ there are no outgoing transitions defined. In $I$'s incarnation $A$, however, a transition is added leading from $A$ to $B$ (line 18). Also, an incoming action $inA(u)$ is added to the transition from $B$ back to $A$ (line 17).

### Adding Preemption Actions

When incarnating a state, preemption actions can be added to the original state definition. Preemption actions can be added to react to the termination of the incarnation's superstate. Adding preemption actions is useful, for example, to include an existing stand-alone program as a sub-phase of a new, more complex program. In our example Prog. 6.4 there are no preemption actions added to the incarnations of either states, since their common superstate $PROG\_4$ is never left.

## 6.2 OSM Language Mapping

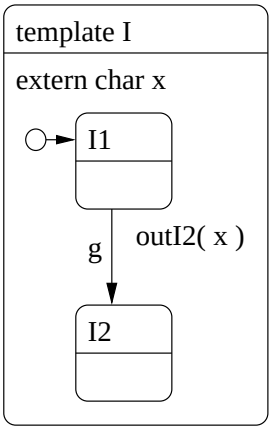
An OSM language mapping defines how OSM specifications are translated into a host language. The host language is also used for specifying actions. Such a mapping is implemented by an OSM compiler. For every host language, there must be at least one language mapping. Yet, there may be multiple mappings focusing on different aspects, such as optimizations of code size, RAM requirements, execution speed, and the like. We have developed a prototypical version of an OSM compiler that uses C as a host language. Hence, we discuss how OSM can be mapped to C in the following sections. The main goal of our compiler prototype is to create executable OSM programs that

- have low and predictable memory requirements and
- are executable on memory-efficient sensor-node operating systems.

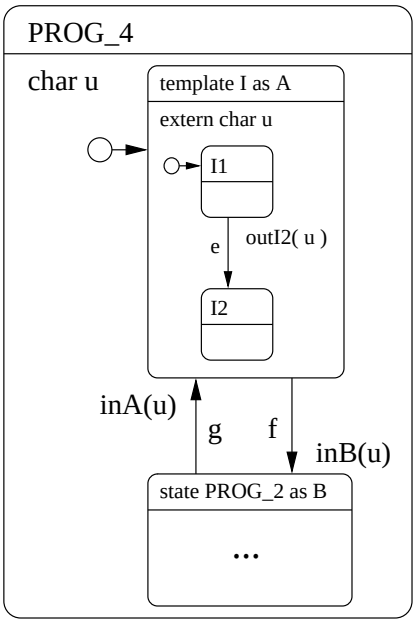
In general, OSM programmers do not need to know the details of how OSM programs are mapped to executable code. What they do need to know, however, is how actions in OSM are mapped to functions of the host language. Programmers need to provide implementations of all actions declared in OSM, and thus need to know their exact signature.

Internally, and typically transparently to the developer, our compiler prototype maps the control structure and state variables of OSM programs separately. *State variables* are mapped to a static in-memory representation using C-language `structs` and `unions`. This mapping requires no dynamic memory management and is independent of the mapping of the control structure. Yet it is memory efficient, as state variables with non-overlapping lifetimes are mapped to overlapping memory regions. Also, the size of the in-memory representation of all state variables is determined at compile time.

Program 6.4: Example for the incarnation of states. *PROG_4* incarnats two states. The fully specified state template *I* (defined in lines 1-9) is incarnated in line 14 as state *A*. And the self-contained OSM program with top-level state *PROG_2* from Fig. 6.2 is incarnated in line 21 as state *B*.




```
1  template state I {
2      extern var char x;
3      extern event g;
4      action outI1( char );
5
6      initial state I1 {
7          g / outI1( x ) -> I2; }
8      state I2 {}
9  }
```




```
10 state PROG_4 {
11     var char u;
12     extern events e,f,g;
13
14     initial state A incarnate I (
15       char u as x,
16       event e as g ) {
17         onEntry: B -> g / inA( u );
18         f / -> B;
19     } // end state A
20
21     state B incarnate PROG_2 (
22       event e as f,
23       event f as e ) {
24         onEntry: A -> f / inB( u );
25         g / -> A;
26     } // end state B
27 }
```

The program's *control structure* is mapped to executable code that manages the internal program state (as opposed to state modeled as state variables) and triggers the execution of actions. Because actions access state variables, the control-structure mapping depends on the mapping of state variables. The generated code closely resembles conventional event-driven program code. It is executable on (slightly modified, yet) memory-efficient system software for event-driven programs.

### 6.2.1 Variable Mapping

A language mapping must define how state variables are allocated in memory, yielding an in-memory representation $R$ of all state variables of an OSM specification. Our main goal is to minimize the memory footprint. The allocation can be defined recursively as follows, where every state of an OSM program has a representation in the C memory mapping.

A single variable is mapped to a memory region $R_{var}$ that is just big enough to hold the variable. A state results in a representation $R_{state}$, which is defined as a sequential record (i.e., a C `struct`) of the representations $R_{var_i}$ of all variables $i$ and the representation of embedded states, if there are any.

The representation of embedded states differs for sequential and parallel compositions. Since a sequential machine can be in only one state at a time, the state variables of different states of the sequential composition can be mapped to overlapping memory regions. Hence, the representation $R_{seq}$ for a sequential state machine is defined as the union (i.e., a C `union`) of the representations $R_{state_i}$ of all states $i$. The states of parallel compositions, however, are always active at the same time, and so are their state variables. Hence, the representation $R_{par}$ of a set of parallel states is defined as a sequential record of the representations $R_{state_i}$ of all contained states $i$.

In the example shown in Program 6.5, the state machine to the left results in the memory layout in the C language shown to the right. If an `int` consists of 2 bytes, then the structure requires 8 bytes. If all variables were global, 12 bytes would be needed. Note that the size of the required memory as well as the memory locations of variables are already known at compile time. With this mapping, no memory management is required at all. Particularly, no dynamic allocations are performed at runtime. Instead, the resulting in-memory representation is statically allocated once (in the code generated from the OSM control structure) and does not change during runtime.

Now we can already define our mapping from OSM actions to function signatures in C. OSM action names translate to C functions of the same name. In the C mapping, state variables can be accessed by their name after dereferencing the variable's associated state in the memory mapping. For example, the state variable $c$ of state $C$ (line 4) maps to `_statePROG_5._par_C_D._stateC.c`. In order to allow their modification in actions, state variables are passed by reference in the invocation. For example, the action `outA(c)` used in line 8 of the OSM program has the following signature in C:

```
outA( *int );
```

Its invocation from the executable code that is generated from OSM control structures is mapped to the following C function call (which is not visible to the programmer):

```
outA( &_statePROG_5._par_C_D._stateC.c );
```

### 6.2.2 Control Structures

Fig. 6.1 suggests that OSM specifications are translated directly into program code of the host language. In our current prototype of the OSM compiler, this

Program 6.5: Example of an OSM program containing several state variables side-by-side with the its variable mapping in the host language C. Every state mapps into a corresponding `union`, containing both the variables of that state plus a representation of the state's substates. Sequential and parallel compositions of substates are represented as a `union` and a `struct` of substate representations, respectively.

```
state PROG_5 {

    state C {
        var int c;

        state A {
            var int a1, a2;
            e / outA(c) -> B;
        } // end state A
        state B {
            var int b1, b2;
            f / -> A;
        } // end state B

    } // end state C
  ||
    state D {
        var int d;
    }

} // end state PROG_5
```

```
struct statePROG_5 {
    struct par_C_D {
        struct stateC {
            int c;
            union seq_A_B {
                struct stateA {
                    int a1, a2;

                } _stateA;
                struct stateB {
                    int b1, b2;

                } _stateB;
            } _seq_A_B;
        } _stateC;

        struct stateD {
            int d;
        } _stateD;
    } _par_C_D;
} _statePROG_5;
```

is only true for the variable mapping. Control structures are first mapped to an intermediate representation in the imperative language Esterel [19, 18, 26]. From this representation, C code is generated by an Esterel compiler.

### The Esterel Synchronous Language

Esterel is a synchronous language (see, for example, [17, 50, 95]) for the specification of reactive systems. Synchronous languages are reactive in that they react to input stimuli (i.e., events) in order to compute output events, typically also changing the internal program state. Synchronous languages are typically used to implement control systems, such as, industry process control, airplane and automobile control, embedded systems, bus interfaces, etc. They are built on the hypothesis that operations take no (real) time, that is, operations are atomic and the output of an operation is synchronous with its input (hence the name synchronous language). Synchronous languages have a discrete model of time where time progresses only on the occurrence of events.

The Esterel language allows to specify concurrent processes, where individ-

ual processes advance on the occurrence of specified events only. In each step, the system considers one or more input events. According to the input, possibly multiple Esterel processes are invoked simultaneously. When invoked, Esterel processes can emit (possibly multiple) external events. They can also invoke procedures. External events and procedures are implemented in Esterel's host-language C. External events are typically used to drive external controller systems while procedures are used to implement complex calculations in the host language C. Programs specified in the Esterel language can be compiled to C and hardware circuits. For the C-code generation there are three different modes, optimized for execution speed, compilation speed, and code size. Pure Esterel programs, that is, programs that do not call any procedures, can also be compiled to hardware-circuit netlists in the Berkeley Logic Interchange Format. Because OSM relies heavily on Esterel's external procedures for the implementation of actions, hardware-circuit generation is not an option.

### Building Programs with Esterel

Esterel programs do not compile to self-contained programs. Rather, they compile to a number of C functions: one principal function and one input function for each possible input event. Additionally, a function stub is generated for each output event and each procedure, which then have to be implement by programmers to complete the program.

The principal C function performs a discrete time step of the Esterel program based on the input events. Input events are passed to the program by calling the corresponding event-input functions before the principal function performs the next discrete step. In the principal function the program's internal state is updated and all procedures are invoked (as well as output events "emitted") by calling the corresponding function stubs. (It should be noted that for programmatically emitting events, OSM does not use Esterel's synchronous event output. Instead, output events are feed into OSM's event queue.) In order to build a full program, programmers also have to supply a main program body that collects event inputs, injects them into the system (via input functions), and then calls Esterel's principal function. This main body resembles the control loop of event-driven systems.

When implementing programs on real systems, the synchronous time model of synchronous languages conflicts with real execution semantics, where operations always require some amount of real time to execute. Therefore, to implement actual programs, programmers must guarantee that input events are only injected into the system after the processing of the previous step has completed. This is typically achieved by buffering input events during the execution of Esterel's principal function, that is, while performing a synchronous step in Esterel. Another approach is to guarantee that at runtime the time interval between any two consecutive input events is longer than the execution time of the principal function. Since the arrival time of external events is typically beyond the control of the programmer, however, this is infeasible for many systems.

The system resulting from the former approach (i.e., queuing) can be seen as an event-driven system, where Esterel's principal function takes the role of a unified event handler for all events. Such a system relies on an underlying

runtime environment to provide an event queue that can hold event sets, and drivers that generate events. The difference between such a system and a traditional event-driven system is that that latter only processes one event at a time while in the former several events can be processed concurrently.

#### Reasons for Building OSM with Esterel

We have chosen to implement the OSM-to-C mapping with Esterel as intermediate language for several reasons. As a synchronous and reactive language, Esterel is well suited for the implementation of state machines. Esterel's programming model and code structure is much closer to OSM as sequential programming languages, such as C. Indeed, Esterel has been used to implement a graphical state-based implementation language called SyncCharts [10, 11]. Our mapping from state-based OSM to imperative Esterel is inspired by this implementation. Also Esterel produces lightweight and system-independent C-code that scales well with the number of states and events and requires no support for multi-tasking. Instead, code generated from Esterel can be integrated well with a modified version of our BTnode system software, which only required moderate effort to support concurrent event sets.

On the practical side, Esterel is immediately available and well documented. Also, the Esterel compiler performs sophisticated analysis of the input sources, thereby detecting logical errors in the intermediate Esterel representation that result from erroneous OSM specifications. This approach relieved us from implementing strict error-detection mechanisms in the prototype compiler ourselves and allowed to focus on the development of the OSM programming model.

### 6.2.3 Mapping OSM Control-Flow to Esterel

In our mapping of OSM, Esterel's principal function performs a discrete time step of the OSM state machine. In each OSM machine step the state transition function accepts the set of concurrent events from OSM's event queue (see Section 5.4.1). In that machine step, the state transition function first invokes all preemption and exit actions of the current state and its substates. Then it performs the state transitions triggered by the set of input events. Finally it invokes all entry actions of the newly assumed states.

Effectively, state-based OSM programs are compiled back into an event-driven program representation where all event types invoke a single action implemented by Esterel's principal function. As a matter of fact, the C code generated from the Esterel representation of OSM programs runs on BTnodes with a slightly modified version of the event-driven BTnode system software. Demonstrating the duality between OSM programs and event-driven programs is the prime goal of the language mapping and an important piece in supporting our thesis. This duality is the basis for showing that state-based programs, such as OSM, do not require more system support than regular event-driven programs. We still need to show, however, that the state-transition function generated from Esterel (i.e., the function that implements an OSM program as event-handler) does not introduce unacceptable overhead. We will do so below.

### Esterel-Code Generation from OSM

The exact details of mapping OSM to Esterel are only of secondary importance. Therefore, and due to space constraints, we will present the mapping by example only. Appendix A.1 shows the OSM program *PROG_PAR* depicted in Fig. 6.2, its mapping to Esterel, and the C code generated by the Esterel compiler. The program is a parallel composition of the OSM state machines presented in Programs 6.1 and 6.2. It has 10 states and contains most features of OSM. Table 6.1 compares the sizes (in terms of lines of code) of the original OSM implementation and the generated C and Esterel mapping code.

| Compilation Unit | # Lines | Comment |
|---|---|---|
| test.osm | 49 | original OSM implementation |
| test.h | 80 | variable mapping (C) |
| test.strl | 307 | control structure mapping (Esterel) |
| test.c | 616 | control structure mapping (C) |

Table 6.1: Number of lines of code for an OSM program and its mappings in C and Esterel.

In this section we demonstrate that the event-driven code generated by OSM does not introduce unacceptable overhead. To this end, we analyze the variable memory and code size of OSM programs. As described previously, the OSM code is first translated into Esterel code, from which C code is generated. Therefore we analyze the object code generated from C code by a C compiler. Since our target platform is the BTnode, we compile for the Atmel AVR 128 microcontroller, the BTnode's CPU. The compiler used is GCC for AVR (avr-gcc), version 3.4.3.

We start by analyzing the Programs 6.1 and 6.2 from the last section. Using these programs as base modules, we the build more complex programs by sequential and parallel composition.

### C-Code Generation from Esterel

The Esterel compiler has several code generation modes and target languages. We use ANSI C in the sorted equations mode. The original C output of the Esterel compiler for program $PROG\_PAR$ depicted in Fig. 6.2 is shown in Appendix A.4.1. The state transition function basically compiles into a long list of boolean equations that operate on a large amount of boolean variables. These variables store the machine's internal program state as well as temporary state during the function's execution in two large array of chars. The representation of boolean values as chars is inherently memory inefficient. Therefore we modify the Esterel-generated C code to use bitfields instead, as exemplified below. Bitfields are a C mechanism for addressing individual bits of a struct. The necessary modifications are made by a Perl script but currently require some human intervention, though the task could be fully automated. Note that these modifications are required only when compiling for the BTnode. Development and initial program testing is typically performed in the BTnode emulation environment on a PC, where size is not an issue.

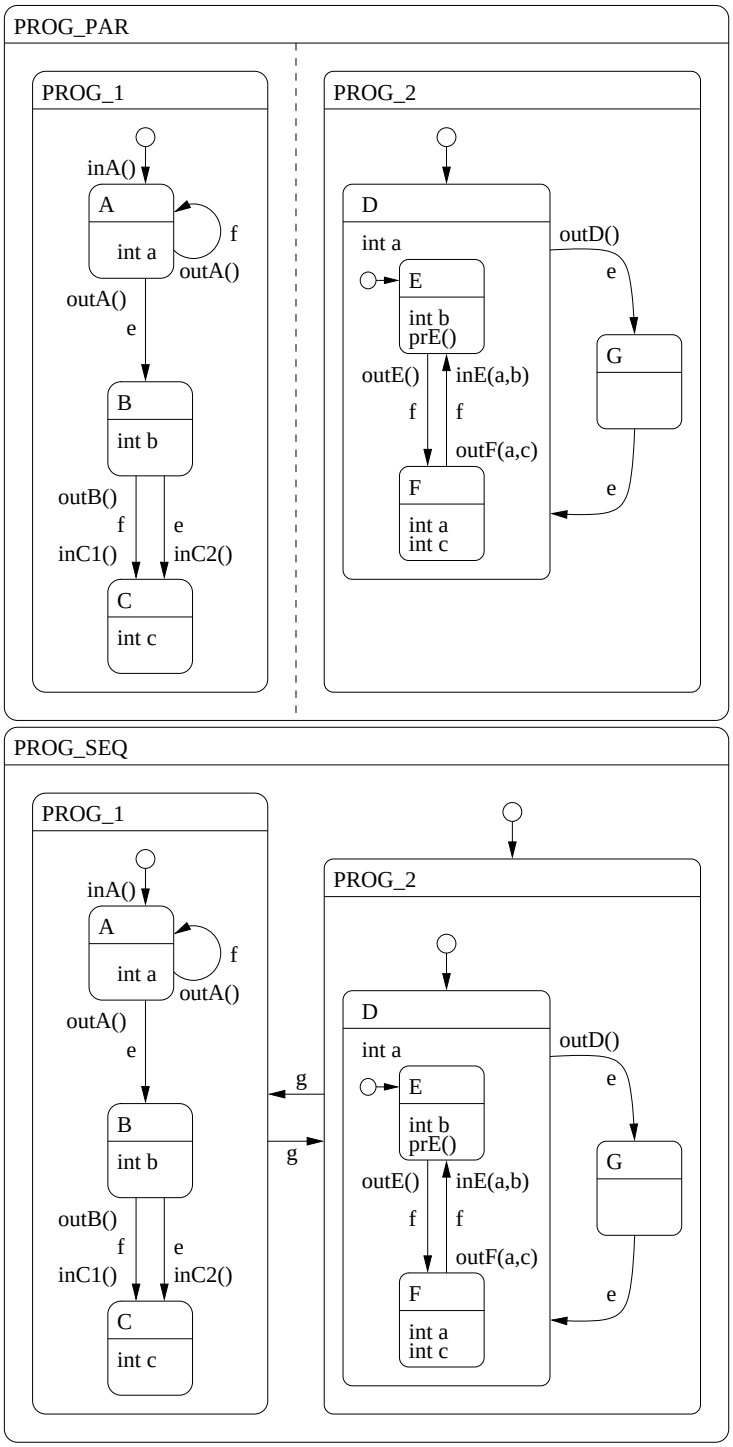



Figure 6.2: Two OSM programs *PROG_PAR* and *PROG_SEQ* composed of the state machines *PROG_1* and *PROG_2* previously presented as Programs 6.1 and 6.2, respectively . In *PROG_PAR*, the two state machines are composed in parallel, while in *PROG_SEQ* they are composed sequentially, introducing a new event $g$ to transition between them.

```
// original representation
char E4;
```

```
// optimized representation
struct {
     unsigned int E0:1;
     unsigned int E1:1;
     unsigned int E2:1;
     unsigned int E3:1;
  } E;
```

The obvious advantage of using bitfiels instead of chars for the storage of boolean values is a reduction of data-memory requirements by a factor of eight. The disadvantage is an increase of program memory by a factor of about 1.8. The size increase stems from the necessity to mask individual bits in the struct. In the disassembly of the modified state-transition function, bit masking shows as additional machine operations in the object code, mainly *immediate AND* operations (i.e., AND with a constant) and register loads. The program-memory size of variable-memory optimized code is shown in Table 6.2.

| **Program** | **Code Size** in bytes | **Memory Size** in bytes | |
|---|---|---|---|
| | Transition Function | Program State | State Variables |
| PROG_PAR | 9240 | 17 | 8 |
| PROG_SEQ | 12472 | 20 | 6 |
| PROG_1 | 3170 | 8 | 2 |
| PROG_2 | 5434 | 10 | 6 |
| D | n/a | 6 | 6 |

Table 6.2: Memory sizes for the states from Fig. 6.2. The programs $PROG\_PAR$ and $PROG\_SEQ$ are concurrently and sequentially composed of program parts $PROG\_1$ and $PROG\_2$, respectively. $D$ is a part of program $PROG\_2$. The first row contains the size of the object code generated for the listed programs. The second row contains the memory requires to internally store their optimized state machine. The third row denotes the size of the state variables in memory.

## 6.3 OSM Compiler

Our OSM-compiler prototype is a three-pass compiler implemented in the Java programming language. In the first pass, the compiler parses the OSM input file and generates an intermediate representation of the program's syntax tree in memory. The parser for the OSM-language is generated by the parser generator CUP from a description of the OSM grammar. (CUP is an acronym for Constructor of Useful Parsers.) CUP serves the same role for Java as the widely known program YACC does for C. CUP takes a grammar with embedded Java code as input (in our case the grammar of OSM), and produces a set of Java classes that implement the actual parser. The parser generated for the OSM language generates an in-memory representation of the OSM syntax tree, that is, it implements the compiler's first pass on an OSM input file. The parser generated by CUP relies on a scanner to tokenize the input of the parser. For our OSM compiler,

the scanner is generated by a lexical analyzer generator called JFLEX (Fast Lexical Analyzer Generator for Java). JFLEX is the Java counterpart of FLEX, a tool generating scanners in the C programming language.

The second pass of the OSM compiler operates on the in-memory syntax tree (created during the first pass) of the OSM input. Concretely, it extends the in-memory representation of interfaces that are only implicitly defined in the input file (such as a complete list of variables visible for each state). It also performs some input verification beyond mere syntactical checks, for example, if all transition targets exist and if all variables used in actions and guards have been defined. However, error checking is mostly delegated to the Esterel compiler on the intermediate Esterel representation of the OSM program. This approach is convenient as we did not need to implement the functionality ourselves, yet can provide some error notifications. On the other hand it is very inconvenient for OSM programmers as they must be able to interpret the somewhat cryptic error messages of the Esterel compiler, which in turn requires to be familiar with the Esterel language.

In the third pass, the Esterel representation of the OSM input is generated according to the language mapping, as described in the previous section. This representation is then compiled to plain C code with the Esterel compiler. The final executable is produced from a C compiler by a generic makefile (see Appendix A.5.2). It compiles and links the three elements that make up an OSM program in the host-language C: (1) the C-language output of the Esterel compiler, representing the program's control flow and structure, (2) the state-variable mappings, which are directly produced as a C header file by the OSM compiler, and (3) the definitions of actions written in C by the programmer.

In the current version the OSM-compiler prototype, some features desirable for routine program development have not been implemented, mostly due to time constraints. For example, the OSM compiler does not support compilation units. Only complete OSM specifications can be processed by the compiler. Note however, that as far as compilation units are concerned, the implementation of actions is independent of the program. Actions are specified in (one or several) C files, which may be compiled separately from the OSM program (see Fig. 6.1).

Also, some OSM features not supported in the current version of the compiler. These features are:

- Incoming transitions with the declaration of a source state (as in `onEntry: A -> e / inA();`): Incoming transitions without the declaration of a source state (such as `onEntry: e / inA();`) work fine. This feature is currently missing, as its implementation in Esterel is rather complex.

- State incarnations and thus adding and renaming actions, transitions and preemption actions to incarnations: We have shown the feasibility and usefulness of state incarnations in a previous implementation of the OSM compiler (presented in [74]). In the re-implementation, this feature was dropped due to time constraints.

- Preemption actions: The realization of preemption actions in the Esterel language is straightforward, yet, would require additional analysis in the

of the second compiler stage. We have dropped this feature due to time constraints.

The entire OSM compiler currently consists of a total of 7750 lines of code, including handwritten and generated Java code (4087 and 2572 lines, respectively), as well as the specifications files for the OSM parser and scanner (707 and 184 lines, respectively). The compiler runs on any Java platform, yet, requires an Esterel compiler [113] for the generation of C code. The Esterel compiler is freely available for the Linux and Solaris operating systems, as well as for Windows with the Cygwin environment installed.

## 6.4 OSM System Software

The OSM system software provides a runtime environment for OSM programs. Because OSM programs are compiled into a basically event-driven program notation, the runtime environment for OSM programs is very similar to regular event-driven programs. In fact, OSM programs run on a slightly modified version of the BTnode system software, our previous, purely event-driven operating system, which we have already described in Sect. 3.5.1. Only small modifications to the control loop, the event queue, and the event representation were necessary in order to support concurrent events. As we pointed out in Sect. 5.4.1, OSM considers the set of events emitted during a single step as concurrent.

In the original BTnode system software, events are implemented as 8-bit values, being able to represent 255 different types of events (event number 0 is reserved). The event queue is implemented as a ringbuffer of individual events. In the revised implementation, events are implemented as 7-bit values, effectively reducing the number of event types that the system can distinguish to 127. The 8th bit is used as a flag to encode the event's assignment to a particular set of concurrent events. Consecutive events in the ringbuffer with the same flag value belong to the same set. Internally, the queue stores two 1-bit values, one for denoting the current flag value for events to be inserted, the other for denoting the current flag value for events to be removed from the queue, respectively.

While the incoming flag value does not change, events inserted into the queue are considered to belong to the same set of events (i.e., their flag is set to the same value). To indicate that a set is complete and the next event to be inserted belongs a different set, the new queue operation *next_input_set* is introduced. Internally, *next_input_set* toggles the value of the flag for inserted events.

Likewise, the queue's *dequeue* operation returns the next individual event from the queue as indicated by the current output flag. The event number 0 is returned if there are no more events for the specified set. To switch to the next set of events the queue operation *next_output_set* is introduced. Additionally, the operation's return value indicates whether there are any events in the queue for the next set or the queue is empty. Both, the *enqueue* and *dequeue* operations retain their signatures.

The control loop also needs to reflect the changes made to the event queue. The fragment below shows the implementation of the new control loop. Instead of dequeuing individual events and directly invoking the associated event handlers, the new version dequeues all events of the current set (line 5) and invokes

the input functions generated by the Esterel compiler (line 6). When all events of the current set have been dequeued, the principal event handler of the OSM program is invoked (line 8). Then the next set of events to be dequeued is activated (line 10). If there are no events pending, the node enters sleep mode (line 11). The node automatically wakes up from sleep mode when a new event is available. Then the process will be repeated.

```
1  event_t event;
2
3  while( true ) {
4
5    while( (event = dequeue()) != 0 )
6      input_to_next_step( event );
7
8    perform_next_step();
9
10   if( next_output_set() == false ) sleep();
11
12 } // end while
```

Note that the *next_input_set* operation is not called from the main loop directly. It is instead called twice at the end of the function performing the next OSM machine step: Once after inserting all events into the queue that have been programmatically emitted by preemption and exit actions. And again after inserting all events that have been programmatically emitted by incoming actions.

## 6.5 Summary

In this chapter we have presented the pieces needed to implement programs based on the concepts of OSM and to compile them into executable code on an actual sensor node. We have presented OSM's specification language that allows to specify sensor-node programs, a compiler to translate OSM program specifications into portable C code, and we have shown how to modify an existing event-driven sensor-node operating system in order to support OSM programs at runtime.

# 7 State-based Programming in Practice

The thesis formulated in the introduction of this dissertation was that the state-based programming model incurs as little overhead as the event-based model, yet allows to program more memory efficiently and allows to specify programs more structured and more modular compared to event-driven programs. In the following sections we will support this thesis.

Before turning to concrete examples, we present an intuitive approach to motivate our claim in Sect. 7.1. We explain how event-driven programs can be formulated as trivial OSM programs containing a single state only. Clearly, such programs do not exhibit the benefits of state-based programming. However, the initially trivial programs are easily refined, increasing their structure, modularity, and memory efficiency. Such refined programs, on the other hand, cannot be translated back in the event-driven model without loosing such benefits again.

In Sect. 7.2 we show how the structure and modularity of sensor-node programs benefits from OSM using a concrete example from the literature. We will compare the reimplementation of a large part of the EnviroTrack middleware in OSM and compare it to the original NesC implementation. We will compare the two implementations with respect to state management, accidental concurrency, and state-bound resource initialization. We show that with OSM manual state management can be removed entirely in favor of an explicit, high-level state notation. As a result the programs are much more concise. In our example, we show that the size of the OSM re-implementation is reduced by 31% (measured in lines of code) with respect to the original nesC implementation.

In Sect. 7.3 we will illustrate how OSM can increase the memory efficiency of programs compared to event-driven programs. Concretely, we will show how state variables foster the reuse of memory for temporary data structures using a small example. The memory savings achievable by using state variables very much depend on the program structure. While very high memory savings can be achieved in theory, we expect savings of 10 to 25% in real-world programs. Sect. 7.4 summarizes this chapter.

## 7.1 An Intuitive Motivation

Traditional event-based programs can be formulated in OSM as depicted in Fig. 7.1. There is a single state $S_0$, which has attached all global variables $var_i$ of the event-based program. For each possible event $e_j$ there is a self transition with an associated action $out_j()$, which has access to $e_j$ and to all state variables of $S_0$. Hence, OSM can be considered a natural extension of event-based programming.

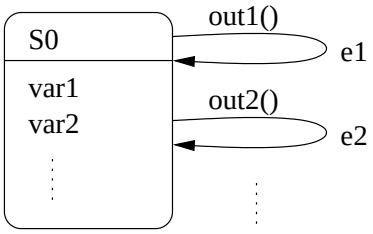



Figure 7.1: A traditional event-based program specified in OSM.

One notable observation is that OSM supports two orthogonal ways to deal with program state: explicit machine states and manual state management using variables. In traditional event-based programming, all program state is expressed via global variables. In pure finite state machines, all program state is expressed as distinct states of the FSM. With OSM, programmers can select a suitable point between those two extremes by using explicit machine states only where this seems appropriate. In particular, a programmer can start with an existing event-based program, "translate" it into a trivial OSM program as shown in Fig. 7.1, and gradually refine it with more states and levels of hierarchy. Naturally, the refined program is more structured.

However, the refined program structure cannot be easily translated back into the event-driven model. To express OSM programs in the event-driven model, the state structure needs to be managed with variables. Manual management has the implications described previously and is a heavy burden for programmers. In OSM, the translation from the state-based to the event-driven program notation, and thus the state management, is automated by the OSM compiler, as described in the previous chapter. Through the translation step we can ensure that the underlying system software is basically the same and thus as efficient as in event-driven systems, yet achive the structural benefits of state-based programming.

Another notable implication of the mapping from the event-driven model to the state-based model is that OSM programs are typically more memory efficient than traditional event-based programs. In the trivial OSM program depicted in Fig. 7.1, as well as in event-driven programs, all variables are active all of the time. Hence, the memory consumption equals the *sum* of the memory footprints of all these variables. In OSM specifications with multiple states, the same set of variables is typically distributed over multiple distinct states. Since only one state of a set of distinct states can be active at a time, the memory consumption of state variables equals the *maximum* of the memory footprints among all distinct states.

Note that in terms of memory consumption, TinyOS programs are like regular event-driven programs and OSM's concept of state variables cannot be applied. Though TinyOS offers components to structure the code, they do so in the functional domain rather than in the time domain. That means that components are always active and thus their variables cannot be mapped to overlapping memory regions.

After this intuitive explanation of the benefits of OSM over pure event-driven programming, we will now discuss the various benefits of OSM using concrete examples in the following sections.

## 7.2 Modular and Well-Structured Program Design

In Chapter 4 we have reviewed the limitations of the event-driven programming model. We identified manual state management as an impediment to writing modular and well-structured program code and as an additional source of errors. In this section we now illustrate how event-driven programs can benefit from a state-based program notation by reimplementing a major part of the EnviroTrack middleware, namely its group management, in OSM. We show that the OSM code is not only much more readable and structured and thus easier to change, it is also significantly shorter then its original implementation in nesC.

We start by giving an overview of the EnviroTrack middleware and explain the original design of its group-management protocol. Then we analyze its original implementation in NesC and present a reimplementation in OSM. Finally we compare both approaches with respect to the number of code lines needed for their specification.

### 7.2.1 EnviroTrack Case Study

We have already briefly mentioned EnviroTrack in Chap. 1. Here we will present a more detailed description. For an exhaustive description, please refer to [6, 7, 23, 54]. EnviroTrack is a framework that supports tracking of mobile targets with a sensor network. In EnviroTrack, nodes collaboratively track a mobile target (an object or phenomenon such as a car or fire) by forming groups of spatially co-located nodes around targets. If a target is detected, the group members collaboratively establish and maintain a so-called context label, one per target. Context labels describe the location and other characteristics of targets and may have user-specified computations associated with them, which operate on the context label's state at runtime. The state is aggregated in the group leader from all group members' sensor values. Context labels are typed to describe different kind of mobile targets. A single node may be member of several groups, each maintaining a its own context label. When the target moves, the context-label group moves with it. Nodes no longer able to detect the target leave the group, while nodes having recently detected the target join. During target movements, EnviroTrack maintains the target's context label, that is, its state.

EnviroTrack is implemented in TinyOS / NesC on Mica2 motes. Its source code is available on the Internet [40]. EnviroTrack consists of 11 NesC components implemented in 45 source files. As discussed in Sect. 3.5.2, NesC components encapsulate state and implementation, and provide an interface for accessing them. The entire implementation consists of 2350 lines of actual source code (i.e., after having removed all empty lines, comments, and debug output).

#### EnviroTrack Group Management

The group management protocol is implemented in the NesC component GroupManagementM.nc. Its implementation accounts for 426 lines, that is, for about 18% of the entire code. We have included a condensed version of GroupManagementM.nc in Appendix B.1. Because for this discussion we are mainly interested in the program's structure, we have removed all code not relevant for

control flow in order to improve its readability. Only statements that are relevant for managing the program's state (i.e., state-keeping, transitions as well as guards in transitions, and state initializations) have been retained. All other statements have been replaced by placeholders (`op`$n$`()` and `[...]`, for operations and for boolean expressions, respectively). A few comments have been added.

To maintain context labels, EnviroTrack employs a distributed group management protocol, which is described in [6]. Per context label, an EnviroTrack sensor node can be in one of four states: `FREE`, `FOLLOWER`, `MEMBER`, and `LEADER`. The state changes only on the occurrence of events: the reception of typed radio messages, timeouts, and the detection as well as the loss of the target (which is computed in a separate module and communicated via *join* and *leave* events). However, a review of the source code revealed that the actual group-management implementation is much more complex. We have reverse engineered the code to find all relevant states, events, and the state transition graph. The result is shown in Fig. 7.2. Compared to the description given in [6], the actual implementation of the protocol has three more states (`NEW_CANDIDATE`, `LEADER_CANDIDATE`, and `RESIGNING_LEADER`) and two more types of radio messages.

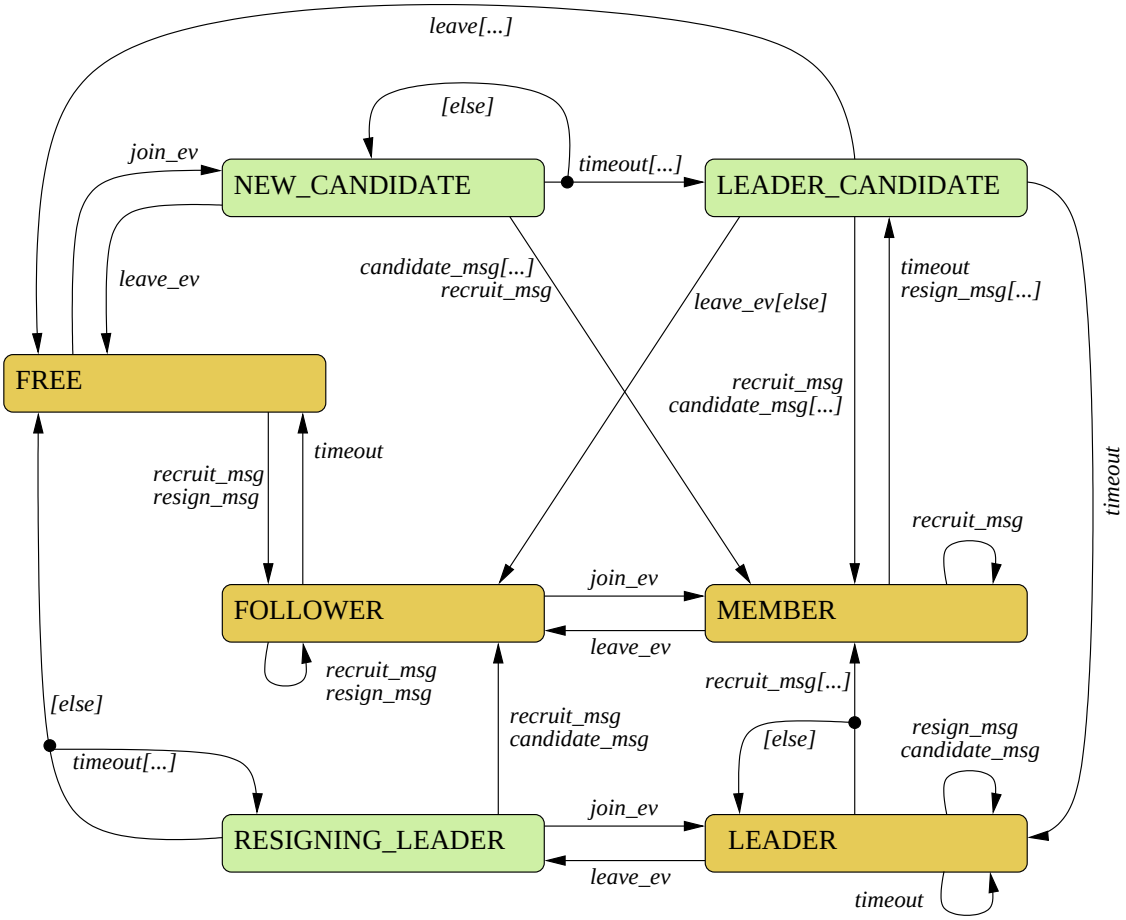



Figure 7.2: Distributed group management of EnviroTrack presented as state machine.

#### Group-Management State Machine

The main states of the group management protocol have the following meaning. A `MEMBER` is a network neighbor of a `LEADER` and is detecting the proximity of the target with its sensors. (The decision whether a target has been detected or lost is made in another component, which signals *join* and *leave* events, respectively.) Members contribute their location estimates and other sensor readings to the target's context label, which is maintained by the group leader. A `FOLLOWER` is a network neighbor of a `LEADER` that does not detect the target itself, that is, followers do not establish a new context label. Followers become members when they start detecting the target. A `FREE` node does not detect the target and is not a `FOLLOWER`. If a free node detects a new target, it establishes a new context label and can eventually become the leader of that group. Finally, a `LEADER` is a `MEMBER` that has been elected out of all members to manage and represent a context label. All members send their location and other sensory data to the `LEADER`, where these locations are aggregated to derive a location estimate of the target. The `LEADER` frequently broadcasts *recruit* messages so that free nodes can detect whether they should become followers, and members know that the leader is still alive. A `FOLLOWER` sets up a timeout to engage in leader election when the leader has not been heard for some specified duration of time, that is, if the timeout expires. Followers not detecting the target any longer become free nodes again.

The actual implementation has tree more states (`NEW_CANDIDATE`, `LEADER_CANDIDATE`, and `RESIGNING_LEADER`) that are used for the coordinated creation of a single leader should several nodes detect a target at (almost) the same time, to handle spurious leaders, and to perform coordinated leader handoff. Also, there are two more messages being sent. The *resign* message is used to indicate that a `LEADER` is no longer detecting the target and is about to resign its role, so that `members` can start leader election in the `LEADER_CANDIDATE` state. Nodes that are candidates for the `LEADER` role (that is, nodes observing the target without previously having heard from a `LEADER`) send a *candidate* message. The *candidate* message is used to prevent the instantiation of several leaders if several nodes have detected the target at almost the same time. In that case several *candidate* message are sent. If a leader candidate receives another *candidate* message, the receiving node becomes a `MEMBER` if the sender has a higher node identifier. We will discuss the NesC implementation of this algorithm and its re-implementation in OSM in the following section.

### 7.2.2 Manual versus Automated State Management

#### NesC Implementation

The implementation of EnviroTrack's distributed group management protocol manages program state manually. The NesC group-management component has a dedicated private variable `status` for keeping the current state of the group-management state machine. (The `status` variable is a field in a larger structure `GMStatus` that holds information relevant to the group management protocol, see Appendix B.1.) The seven states of the state machine are mod-

eled explicitly as a NesC-language enumeration. Transitions are modeled by assigning one of the enumeration constants to the state-keeping variable. (We use the term "state-keeping variable" as opposed to "state variable" in order to distinguish them from OSM's concept of state variables.) The code resembles the general pattern of event-driven applications as shown in Program 4.2. As such, it is very structured for an event-driven program.

Program 7.1 is an excerpt of the original NesC code representing a state transition (see also Appendix B.1, lines 159-164). The `MEMBER` state is entered from the `FOLLOWER` state through a transition triggered by a *join* event (line 1). In order to increase the readability of the code, `op7()` (line 5) serves as a placeholder for the actual action to be performed on the occurrence of the *join* event in state `FOLLOWER`. Then the transition's target state is assigned in line 6. Finally, the timer of the target state `MEMBER` is initialized in line 7. We will discuss initialization of state-bound resources, such as timers, in greater detail below.

Program 7.1: NesC code fragment implementing the transition from `FOLLOWER` to `MEMBER` upon a *join* event. The code also includes the timer initialization of the target state (line 7).

```
1   command result_t GroupManagement.join() {
2      switch( _GMStatus.status ) {
3      case ...: // [...]
4      case FOLLOWER:
5         op7();
6         _GMStatus.status = MEMBER;
7         _generalTimer = wait_receive();
8         break;
9      case ...: // [...]
10     default:  // [...]
11     }
12  }
```

### OSM Implementation

We have reimplemented EnviroTrack's group management state machine in OSM, as shown in Appendix B.2. A small excerpt of the OSM code is shown in Prog. 7.2, which is the OSM equivalent to the NesC fragment in Prog. 7.1. Instead of using a state-keeping variable, states are specified explicitly using OSM states (indicated by the `state` keyword). Transitions are specified explicitly within their source state, making switch/case statements obsolete. Unlike in event-driven programs, control-flow information and implementation are clearly separated in OSM programs. OSM code only contains the control flow of the group management protocol. It does not contain any program logic. The actual program is implemented in the C language in separate files.

Program 7.2: OSM code fragment implementing the transition from `FOLLOWER` to `MEMBER` upon a *join* event. The code also includes the timer initialization of the target state (line 6).

```
1 state FOLLOWER {
2   join_ev / op7() -> MEMBER;
3   // [...]
4 }
5 state MEMBER {
6   onEntry: / reset_timer( wait_receive() );
7   // [...]
8 }
```

### Comparison

There are two main differences between the event-driven and the state-based programming model regarding program structure. Firstly, the order of structuring devices—states and events—is reversed. In the event-driven model, the first-level structuring device is events, or rather event handlers. In the event handlers, control flow is then manually multiplexed (with `switch` statements) according to the (manually coded) program state. In the state-based model of OSM, on the other hand, programs are essentially constructed from hierarchically structured states, in which the control flow is multiplexed according to events. Allowing programmers to specify their programs in terms of freely arrangeable hierarchical states offers more freedom and a better way of structuring, rather than being confined to a few predetermined event handlers. OSM program code can be divided into multiple functional units, each implemented in a state or state hierarchy. Because of the clearly defined interfaces of states, their implementation can be easily replaced by another implementation with the same interface. In general, using state machines is a more natural way to describe programs (see Sect. 4.2).

The second difference between both models is that program state in OSM is a first-class abstraction and state management can thus be automated. The high-level state notation is much more concise compared to the code in the event-driven model, which is lacking such an abstraction and thus state has to be managed manually. From the high-level description of the program's state structure, code for multiplexing the control flow is generated automatically and transparently for the programmer. The programmer is saved from having to manually code the control flow. The amount of code that can be saved is significant, as we will show for our EnviroTrack example.

The code structure of the EnviroTrack group-management implementation in NesC (see Appendix B.1) is clearly dominated by manual stack management necessitated by the event-driven programming model. As can be seen from Tab. 7.1, out of the 426 code lines of the original NesC implementation, 187 lines (i.e., about 44%) are dedicated to manual state management. The remaining 239 lines mainly contain the actual program logic (but also some declarations and interface declarations). That is, out of 9 lines of code, 4 lines are required

| **EnviroTrack Group management** | **Size** in Lines of Code | | **Size Reduction** |
|---|---|---|---|
| | NesC | OSM | with OSM |
| Flow control (state management) | 187 (44%) | 56 (19%) | 70% |
| Program logic | 239 (56%) | 239 (81%) | n/a |
| Total | 426 (100%) | 295 (100%) | 31% |

Table 7.1: Comparison of program sizes for the implementation of EnviroTrack's group-management protocol in NesC and OSM.

to explicitly specify the program's control flow, the remaining 5 lines implement the actual program logic.

On the other hand, an equivalent program in OSM requires only 56 lines of code, that is, less than one third of the original NesC code size. In OSM, the fraction of code for managing the control flow (56 lines) against the entire program (56+239=295 lines) is down to less then 20%. That is, there is only a single line of code required to express the programs control flow to every 4 lines of program logic (which is implemented in separate files). The results of this comparison are summarized in Tab. 7.1. Note that in NesC programs the implementation of the actual program logic does not use any NesC-specific features; just like in OSM the actual program code is specified using plain C statements and is thus identical in both programming frameworks.

In this section we have looked at manual state management in general. In the next section we look at special patterns of manual state management that regularly appear in typical sensor-node programs, namely state-based resource initialization and avoiding accidental concurrency.

### 7.2.3 Resource Initialization in Context

In our analysis of sensor-node programs in Sect. 4.2 we stated that the computations within a program state often utilize a well defined set of resources. Typically these resources are allocated as well as initialized when entering the program state, and released again when leaving the state. To efficiently specify this resource allocation and release, OSM offers incoming and outgoing actions (respectively). Typical examples in sensor networks are the state-based initialization and use of hardware resources, such as sensors and emitters, but also transceivers, external memories, timers, and so on. A special case of a state-bound resource is memory that is only used in the context of a state, that is, state variables. We will discuss how state variables are used in practice in Sect. 7.3. As resource initialization and release are bound to state changes, these operations are closely connected to state management. In the following we will show how the state-based approach of OSM helps improving the code structure with regard to resource initialization. To do so, we will again use the EnviroTrack example.

#### Resource Initialization in EnviroTrack

In our EnviroTrack example, all states (except FREE) use a timer. The timers in the EnviroTrack group-management protocol always trigger actions specific to

the current state. These actions are either state changes or recurring computational actions within the current state. Therefore, we like to think that conceptually every state uses an individual timer, though both the original NesC and the OSM implementations only use a single hardware timer for all states. The individual timers are always initialized in transitions, either when entering a new state through a transition, or after triggering a recurring action within the current state, that is, in a self transition. Table 7.2 lists the values that are used to initialize the timer in the EnviroTrack example based on the transition parameters, namely the target and source states as well as the trigger event. Using this example, we will now explain how state-bound resources are initialized in both the event-driven and the state-based programming models and then compare both approaches.

| **Target State** | **Source State** | **Trigger** | **Line No.** | | **Timeout Value** |
|---|---|---|---|---|---|
| | | | NesC | OSM | |
| NEW_CANDIDATE | FREE | join | 154 | 27 | wait_random() |
| | *NEW_CANDIDATE* | *timeout[else]* | 106 | 29 | wait_recruit() |
| LEADER_CANDIDATE | NEW_CANDIDATE | timeout[...] | 111 | 36 | wait_random() |
| | MEMBER | timeout | 100 | 36 | |
| | MEMBER | resign[...] | 269 | 36 | |
| FOLLOWER | FREE | recruit OR resign | 248 | 12 | wait_threshold() |
| | LEADER_CANDIDATE | leave[else] | 200 | 12 | |
| | *FOLLOWER* | *recruit OR resign* | 256 | 12 | |
| | MEMBER | leave | 186 | 12 | |
| | RESIGNING_LEADER | recruit OR candidate | 337 | 12 | |
| MEMBER | NEW_CANDIDATE | recruit | 279 | 19 | wait_receive() |
| | NEW_CANDIDATE | candidat[...] | 284 | 19 | |
| | LEADER_CANDIDATE | recruit[...] | 295 | 19 | |
| | LEADER_CANDIDATE | candidate[...] | 300 | 19 | |
| | FOLLOWER | join | 161 | 19 | |
| | *MEMBER* | *recruit* | 264 | 19 | |
| | LEADER | recruit[...] | 313 | 19 | |
| LEADER | LEADER_CANDIDATE | timeout | 120 | 45 | wait_random() |
| | RESIGNING_LEADER | join | 169 | 46 | wait_recruit() |
| | *LEADER* | *timeout* | 126 | 47 | — ″ — |
| | *LEADER* | *recruit[else]* | 317 | 47 | — ″ — |
| | *LEADER* | *resign OR candidate* | 325 | 47 | — ″ — |
| RESIGNING_LEADER | *RESIGNING_LEADER* | *timeout[...]* | 135 | 56 | wait_recruit() |
| | LEADER | leave | 209 | 56 | |

Table 7.2: Timeouts in the EnviroTrack group-management protocol are initialized as part of transitions and predominantly depend on the target state only. The source states and triggers of self transitions are set in italics. Line numbers refer to the source code given in Appendices B.1 and B.2 (for the NesC and OSM version, respectively). The actual timeout value is expressed as the return value of a function.

### Original NesC Implementation

In the NesC implementation of the EnviroTrack group-management protocol, a single timer is used to implement timeouts in all states using a count-down approach. The timer is switched on when the program starts and is never switched off. It fires at a regular interval that never changes. The corresponding timeout

handler `fireHeartBeat()` of the NesC implementation is shown in line 80 et seqq. in Appendix B.1. An excerpt of the timeout handler is shown below. The timeout handler counts down a integer variable (line 5), except when in the `FREE` state (line 2). When the count-down variable reaches zero (line 7), an action is invoked, a transition is triggered, or both. Additionally, the count-down variable is reset. The code fragment below implements the transition from `FOLLOWER` to `FREE` in the lines 9-13 (also see the state-transition diagram in Fig. 7.2). This transition is triggered when neither the leader has been heard nor the target has been detected until the timeout fires.

```
1   command result_t GroupManagement.fireHeartBeat() {
2      if( _GMStatus.status == FREE )
3         return SUCCESS;
4      if( _generalTimer > 0 )
5         _generalTimer--;
6
7      if( _generalTimer <= 0 ) {
8
9         switch( _GMStatus.status ) {
10        case FOLLOWER: {
11           initGMStatus();   // transition to FREE
12           break;
13        } // end case FOLLOWER
14
15        ...
16     } // end switch
17   } // end if
18   return SUCCESS;
19 } // end command
```

As we mentioned previously, timers are initialized in transitions. Concretely, a timer of a particular state is initialized in transitions that have that state as a target state. In our example, the timer of the `FOLLOWER` state is initialized in transitions from the states `FREE`, `LEADER_CANDIDATE`, `MEMBER`, and `RESIGNING_LEADER`, as well as a self transition. As these transitions are triggered by several events, the timer initializations occur in the several event handlers. The code fragment below shows the initialization of the count-down timer in the transition from the `MEMBER` state (lines 3-8) as well as in the self transition (lines 17-23). The line numbers of all timer initializations in the NesC code (see Appendix B.1) is given in Table 7.2. (Please note that the line numbers given in the code fragments do not correspond to the line numbers in the appendices.)

```
1  command result_t GroupManagement.leave() {
2     switch( _GMStatus.status ) {
3     case MEMBER: {
4        op9();
5        _GMStatus.status = FOLLOWER;
6        _generalTimer = wait_threshold();
7        break;
8     } // end case MEMBER
9     // ...
```

```
10      } // end switch
11    } // end command
12
13    task void ProcessRecuritMessage() {
14       GMPacket* RxBuffer;
15       // ...
16       switch( _GMStatus.status ) {
17       case FOLLOWER: { // self transition
18          if( (RxBuffer->type==RECRUIT) || (RxBuffer->type==RESIGN) ) {
19             op14();
20             _generalTimer = wait_threshold();
21          }
22          break;
23       } // end case FOLLOWER
24       // ...
25       } // end switch
26    } // end task
```

### OSM Re-Implementation

In our OSM re-implementation (see Appendix B.1), we also use a single hardware timer. However, instead of using a fixed-rate timer and a count-down variable, we directly initialize the timer to the desired timeout value. This reflects the state-centric view in that an individual timer exists for every program state and saves us from having a global event handler to count down the variable. So instead of resetting a global count-down variable (as `_generalTimer` in the NesC code), we initialize a state-specific timer. In OSM, the timers are initialized upon state entry in the incoming transition using the `reset_timer()` function, as shown in the code fragment below. However, for the simplicity and efficiency of the implementation, we sill use a single timer. This approach is more natural to the state-based approach of OSM and yields the same results as the NesC implementation.

The code fragment below shows the timer initializations for the `FOLLOWER` state in the OSM initialization. The incoming transition denoted by the `onEntry keyword` neither specify the transition's source state nor its trigger. Therefore, the action is always performed when the state is entered.

```
1  state FOLLOWER {
2    onEntry:                    / reset_timer( wait_threshold() );
3    join                        / op7()  -> MEMBER;
4    timeout                     / op11() -> FREE;
5    recruit_msg OR resign_msg   / op14() -> self; // invokes onEntry actions
6  }
```

### Comparison

When analyzing the code of both implementations with respect to the number of initializations required, as summarized in Table 7.2, we find that in the NesC code there are 24 lines in which a timeout initialized compared to 9 individual lines in OMS. That is, multiple initializations in a particular state in

NesC can be condensed to a single line in OSM, if the initialization procedure is identical. For example, when entering the `FOLLOWER` state (from another state or a self transition) the timer is always initialized to a value returned by the `wait_threshold()` function. This can be written in a single line, as in the code fragment above (line 2).

But even when initializations cannot be condensed because initializations differ (e.g., the timeout value varies with the transition through which the target state is entered instead of being strictly state-specific), OSM has a clear benefit. While in this case NesC initializations for a single target state are scattered throughout much of the program, they are neatly co-located in OSM. For example, the initializations for the `LEADER` state are performed in lines 120, 126, 168, 316, and 324 in the NesC code, while in OSM they are performed in lines 45, 46, and 47.

In real-world sensor-node programs, the actual initialization procedures (and values, as in our timer example) very often depend exclusively on the target state of the transition. In OSM, these initializations can be easily modeled as incoming transitions. If the initialization code is equal for all transitions to a certain state, they can be condensed to a single line as in `FOLLOWER` state already mentioned above, like this:

```
onEntry:  / reset_timer( wait_threshold() );.
```

Incoming transitions are specified in the scope of a state description (using the `onEntry` keyword). Therefore, all initialization code is co-located, even if the concrete initialization depends on transition parameters such as source state, trigger events, and guards. For example, the timer initializations in the leader state are performed as shown in the code fragment below in OSM. In event-driven programs, however, the initializations of a resource belonging to a particular state are scattered throughout the program together with the event handlers that implements transitions to that state.

```
1 state LEADER {
2   // incoming transitions:
3   onEntry: LEADERCANDIDATE -> / reset_timer( wait_random() );
4   onEntry: RESIGNINGLEADER -> / reset_timer( wait_recruit() );
5   onEntry: self ->            / reset_timer( wait_recruit() );
6   // self transitions:
7   timeout                     / send_recruit_msg() -> self;
8   // more transitions follow...
9 }
```

In sensor-node programs timers are also often used to implement recurring operations without state changes, such as finding network neighbors, sampling, etc. In our example, the recruit message is send regularly in the `LEADER` state. In OSM, this can be modeled with an explicit self transition as shown in line 7 of the previous code fragment. The self transition then automatically triggers the appropriate `onEntry` action, that is, line 5 in the above code. Where a self transition does not seem adequate, a regular action without transition can be used instead, as shown below. Though this requires multiple lines of code even for identical initializations, the code is still part of the definition of a single state.

```
timeout / reset_timer( wait_threshold() );
```

### 7.2.4 Avoiding Accidental Concurrency

Another special case of manual state management is program code to avoid accidental concurrency. In Sect. 4.1 we stated that accidental concurrency is a source of error and may lead to corrupted data and deadlock. Accidental concurrency may occur when a non-reentrant cascade of events is triggered again before the previous cascade has completed.

Most event-driven programs may suffer from accidental concurrency and special measures must be taken by programmers to avoid it. A common pattern used throughout the many event-driven programs that we have analyzed is to simply ignore events that would restart a non-reentrant cascade of events. However, from our own experience we know that potentially harmful situations in the code are not easily identified or that programmers quite often simply forget to implement countermeasures.

On the other hand, most OSM programs are "by design" not susceptible to accidental concurrency. In the following we will provide two examples of event-driven programs where measures against accidental concurrency have been taken. The first of the two examples is the Surge application, which we have already discussed in Section 4.2.1. We will use this small example to explain the measures typically taken by programmers to avoid accidental concurrency and why typically no measures are needed in OSM. The second example is again EnviroTrack. We will analyze the entire EnviroTrack code in order to quantify the overhead of countermeasures against accidental concurrency.

Our first example is the Surge application taken form the publication presenting the NesC language [45]. Triggered by a regular timeout, the Surge program (as shown in Prog. 7.3) samples a sensor and sends the sample off to a predetermined node.

The order of actions (i.e., NesC tasks and events) in the code of Prog. 7.3 suggest a linear order of execution that does not exist. Particularly, the timeout (line 6) may fire again before sending actually succeeded with an *sendDone* event (line 22). Therefore, the programmer of Surge choose to introduce the `busy` flag (line 3) to indicate that the cascade has not been completed (line 9) and that subsequent timeouts need to be ignored until the flag is reset (line 23).

In OSM state machines, all events that are to trigger a transition or an action within a particular state need to be specified explicitly. As OSM programmers would typically explicitly model sequences of events as a state machine, unexpected events cannot accidentally trigger a cascade of events. Program. 7.4 shows the implementation of Surge in OSM, which does not require an explicit `busy` flag. The OSM state machine only reacts to a *timeout* event while in the `IDLE` state (line 3). The program can only be restarted after again entering the `IDLE` state, that is, after the *sendDone* event has occurred (line 19). Intermittent *timeout* events are ignored. Alternatively, an error handler could be added to the `BUSY` state after line 19, like this:

```
timeout / error_handler();.
```

In the previous example, there was only one potential case of accidential concurrency, which was countered by a single flag that was accessed twice. In real-world programs, however, more effort is required to guard against accidental concurrency. For example, in the EnviroTrack group-management code

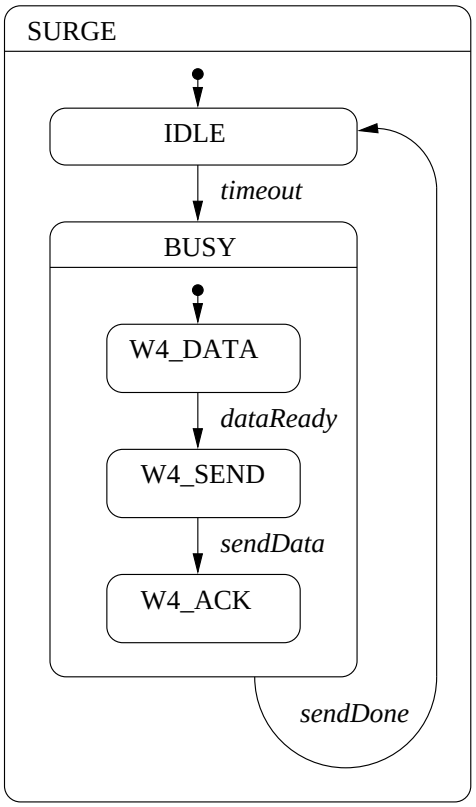
Program 7.3: NesC code of the Surge program from [45] (slightly simplified version).

```
1 module SurgeM { ... }
2 implementation {
3   bool busy;
4   uint16_t sample;
5
6   event result_t Timer.fired() {
7     if( busy ) return FAIL;
8     call ADC.getData();
9     busy = TRUE;
10    return SUCCESS;
11  }
12  event result_t ADC.dataReady( uint16_t data ) {
13    sample = data;
14    post sendData();
15    return SUCCESS;
16  }
17  task void sendData() {
18    adcPacket.data = sample;
19    call Send.send( &adcPacket, sizeof adcPacket.data );
20    return SUCCESS;
21  }
22  event result_t Send.sendDone( TOS_Msg *msg, result_t success ) {
23    busy = false;
24    return SUCCESS;
25  }
26 }
```

2 boolean flags are used, which are accessed in 16 of the 426 lines, that is, almost 4% of the group-management code. In the entire EnviroTrack implementation there are 9 boolean flags, which are accessed in 58 lines of code. That is, measures against accidental concurrency cost 9 bytes of memory and account for 3% of the entire code.

In the previous example, most of the flags are introduced in the processing of radio messages. Several components that process radio messages have a single buffer for incoming messages only. Thus, a received message has to be processed completely before a new one can be accepted into the buffer. Particularly, the previous message must not be overwritten while being processed. Similar to the previous Surge example, EnviroTrack simply drops radio messages that are received while still processing a previous message. The same also applies to outgoing messages.

Program 7.4: Reimplementation of SURGE in OSM. A graphical representation of the OSM state machine is depicted to the left.




```
1  state SURGE {
2    initial state IDLE {
3      timeout/ ADC_getData() -> BUSY;
4    }
5    state BUSY {
6      uint16_t sample;
7
8      initial state W4_DATA {
9         dataReady/
10           sample=dataReady.data,
11           post( sendData ) -> W4_SEND;
12     }
13     state W4_SEND {
14       sendData/ send(sample)->W4_ACK;
15     }
16     state W4_ACK {
17       // do nothing
18     }
19     sendDone/ -> IDLE;
20   } // end BUSY
21 } // end SURGE
```

## 7.3 Memory-Efficient Programs

In the previous section we have shown that partitioning sensor-node programs in the time domain through explicitly modeled states notation benefits modularity and structure of the program code. In this section we show that OSM's concept of state variables—variables the lifetime of which are associated with states—can also benefit memory efficiency.

The basic idea of saving memory is to allocate memory only for those variables, that are required in the currently active state of the program. When the program state changes, the memory used by variables of the old program state is automatically reused for the variables of the new state. In our state-based approach, states not only serve as a structuring mechanism for the program, they also serve as a scoping and lifetime qualifier for variables. Regular event-driven programming models lack this abstraction and thus force programmers to use global variables with global lifetime, as described in Sect. 4.1.

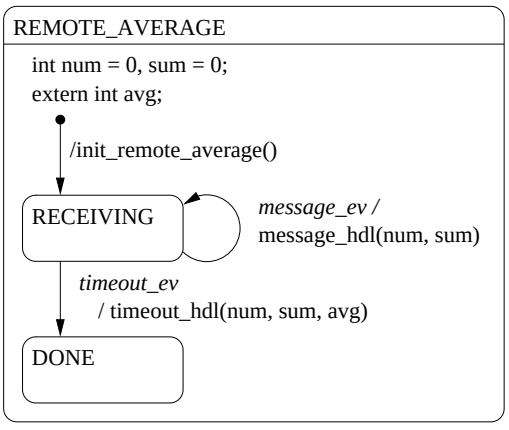
Using an example based on the program to calculate the average of remote temperature sensors from Sect. 4.1, we now show how the state-based programming model can be used in practice to save memory by storing temporary data in state variables. Then we answer the question how this technique is generally applicable to sensor-node programming and how much memory can be saved in real-world programs.

### 7.3.1 Example

In the previous sections of this chapter we have always referred to the EnviroTrack group-management protocol for comparing the event-driven and the state-based programming approaches. This example, however, is not suitable to demonstrate OSM's memory efficiency. While the protocol can be nicely specified as a set of distinct states, individual states do not have temporary data attached exclusively to them. (However, they have another resource associated to them, namely timers, which we have discussed in the section about accidental concurrency.)

Therefore, we take up the small toy example from Sect. 4.1 again, which we used to explain the shortcomings of event-driven programming. The original, event-driven program (see Prog. 4.1) calculates the average of temperature values from remote sensors in the vicinity. Program 7.5 depicts the equivalent program in OSM as the superstate `REMOTE_AVERAGE`. Just as the original version of the program, the OSM code of `REMOTE_AVERAGE` has two variables `num` and `sum`, which store the number of remote samples received and the sum of all temperature values, respectively. These variables are temporary in nature, as they are only required until the average is computed when the timeout occurs. In the original, event-driven program, these variables are global, thus taking up memory even when they are no longer needed. In OSM, on the other hand, the variables are modeled as state variables. The variables memory are automatically reused by the OSM compiler for state variables of subsequent states.

Program 7.5: OSM equivalent to the event-driven Program 4.1. In this version, the variables `num` and `sum` are modeled as state variables instead of global variables.

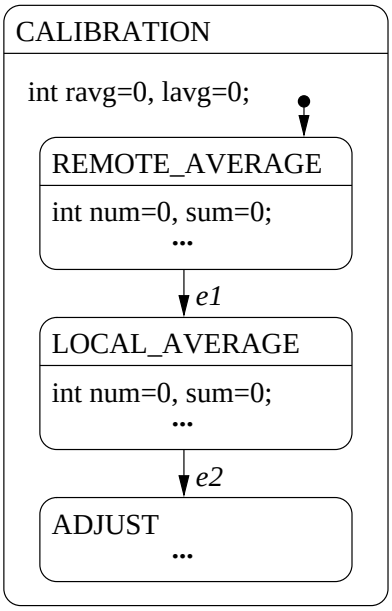



```
1  state REMOTE_AVERAGE {
2    var int num = 0, sum = 0;
3    extern var int avg;
4    onEntry:/init_remote_average();
5    initial state RECEIVING {
6      message_ev /
7        message_hdl(num, sum);
8      timeout_ev /
9        timeout_hdl(num, sum, avg )
10       -> DONE;
11   }
12   state DONE {}
13 }
```

Program 7.6 shows an example where `REMOTE_AVERAGE` is used in conjuncton with two other states. In this program, the node's local sensor is calibrated based on the average of remote temperatures (as calculated by `REMOTE_AVERAGE`) and the average of a series of local temperature readings. The local average is computed in the `LOCAL_AVERAGE` state. Its implementation (not shown here) is very similar to that of `REMOTE_AVERAGE`. Particularly,

it also has two state variables `num` and `sum` with the same meaning. Only instead of collecting temperature values from remote message events, the temperature values are collected from local sensor events. The actual calibration is performed in `ADJUST`. We make no assumption about the implementation of this state.

Program 7.6: OSM program for calibrating a local sensor using the averages of remote and local sensors.




```
14 state LOCAL_AVERAGE { /*...*/ }
15
16 state CALIBRATION {
17   var int ravg = 0, lavg = 0;
18   initial state
19     incarnate REMOTE_AVERAGE (
20       int ravg as avg ) {
21     e1 / -> LOCAL_AVERAGE;
22   }
23   initial state
24     incarnate LOCAL_AVERAGE (
25       int lavg as avg ) {
26     e2 / -> ADJUST;
27   }
28   state ADJUST { /*...*/ }
29 }
```

In this simple yet realistic example, the two subsequent states `REMOTE_AVERAGE` and `LOCAL_AVERAGE` both have local data stored in state variables. Because the states are composed sequentially, their state variables are mapped to an overlapping memory region, effectively saving memory. In this particular case the state variables of both states are of the exact same types and sizes. Thus the state variables of the subsequent state would be mapped to the exact memory locations of those of the previous state. Therefore, the memory saved (with respect to the event-driven approach, where all variables are global) is half of the total memory, that is, the size of two integers, typically 4 bytes on 8-bit microcontrollers.

On the other hand, the given OSM implementation introduces two additional variables, namely `ravg` and `lavg`. They are used to store the remote and local averages until the actual calibration has completed. Indeed, using state variables to share data among substates is a common pattern in OSM. The shared data typically represents results of computations that are performed in one state and need to be accessed in subsequent states, as in the previous example. The requirement of additional variables is a direct consequence of the limited scope and limited lifetime of state variables. They would not be required if all variables where global. Therefore, one could object that those additional variables would cancel out the memory savings gained by state variables with respect to the event-driven approach. In general, however, the temporary data required to compute a result is larger than the result itself. Therefore, the saving typically

only reduced by a small amount. Furthermore, `CALIBRATION` may be an only transient state itself. Then its (as well as its substate's) memory resources would also reused once the state is left. Since `CALIBRATION` does not compute a result required in a subsequent state, no additional variables are needed.

Since the OSM compiler compiles OSM program specifications back to regular event-driven code, one could also object that the mechanism of mapping variables to overlapping memory regions could also be used manually in the plain event-driven model. Though possible in principle, programmers would not only have to manage the program's state machine manually. They would also have to manually map the states' variables to the complex memory representation, such as the one shown in Prog. 6.5. Even minor changes to states high in the hierarchy, such as the addition or removal of a layer in the hierarchy, lead to major modifications of the memory layout. Consequentially, access to elements of the structure would also change with the structure, requiring changes throughout the program. Therefore we think that our state-variable approach is only manageable with the help of a compiler.

### 7.3.2 General Applicability and Efficiency

Now two question arises: are state variables applicable to real-world sensor-node programs? And, how much memory savings can we expect? We cannot answer these questions conclusively, as extensive application experience is missing. However, we have several indications from our own work with sensor-node programming as well as from descriptions of sensor-node algorithms in the literature that state variables are indeed useful.

#### General Applicability of State Variables

Probably the most-generally applicable example where sensor-node programs are structured into discrete phases of operation is initialization and setup versus regular operation of a sensor node. Typically sensor nodes need some form of setup phase before they can engage in their regular operation mode. Often longer phases of regular operation regularly alternate with short setup phase. Both phases typically require some amount of temporary data that can be released once the phase is complete. Setup may involve assigning roles to individual nodes of the network [56, 98], setting up a network topology (such as spanning trees [84]), performing sensor calibration, and so on. Once setup is complete, the node engages in regular operation, which typically involves tasks like sampling local sensors, aggregating data, and forwarding messages. Regular operation can also mean to perform one out of several roles that has been assigned during setup. For the setup phase, temporary data may contain a list of neighboring nodes with an associated measure of link quality, node features, as well as other properties of the node for role assignment and topology setup. Or they may contain a series of local and remote sensor measurements for calibration (as in our previous example). For regular operation, such variables may contain aggregation data, messages to be forwarded, and so on.

### Effective Memory Savings

In theory, state variables can lead to very high memory savings. As we explained earlier, the memory consumption of state variables equals the maximum of the memory of variable footprints among all distinct states. The memory consumption of the same data stored in global variables equals the sum of the memory footprints. For example, in a program with 5 sequential states each using the same amount of temporary data, state variables only occupy $1/5$-th of the memory of global variables, yielding savings of 80%.

However, the effective savings that can be achieved in real-world programs will be significantly lower because of three main reasons. Firstly, often at least some fraction of a program's data is not of temporary nature but is indeed required throughout the entire runtime of the software. In OSM, such data would be modeled as state variables associated to the top-level state, making them effectively global. With the state-variable approach, no memory can be saved on global data.

Secondly, we found that sometimes the most memory efficient state-machine representation of a given algorithm may not be the most practical or the most elegant. For the purpose of saving memory, a state machine with several levels of hierarchy is desirable. On the other hand, additional levels of hierarchy can complicate the OSM code, particularly when it leads to transitions crossing hierarchy levels. As a consequence, programmers may not always want to exploit the full savings possible with state variables.

The final and perhaps most important reason is that in typical programs a few states exceed others in terms of memory consumption by far. It is these dominant states that determine the total memory consumption of an OSM state-machine program. Examples are phases during which high-bandwidth sensor data (such as audio and acceleration) is buffered and processed (e.g., [12, 117]) or phases during which multiple network messages are aggregated (e.g., [84]). State variables do not help reducing the memory requirements of memory-intensive states. For these states, programmers still have the responsibility to use the available resources economically and to fine tune the program accordingly.

The data state (i.e., memory) used by each of the non-dominant states, however, "falls into" the large chunk of memory assigned to the dominant state and is thus effectively saved. A positive side effect of this mechanism is that in most states (i.e., the ones that are less memory intensive) programmers can use variables much more liberally, as long as the total per-state memory is less than the memory of the dominant state. Since the per-state memory requirements are calculated at compile time, programmers get an early warning when they have been using memory too liberally.

Due to the reasons explained above we consider savings in the range of 10% to 25% to be realistic for most programs, even without fine tuning for memory efficiency. The savings that can be effectively achieved by switching from an event-driven programming framework to state-based OSM, however, largely depends on the deployed algorithms as well as the individual programming style.

## 7.4 Summary

In this chapter we have illustrated how the state-based programming approach of OSM compares to the "traditional" event-driven approach. In particular, we have shown how largely automating state and stack management in OSM benefits programmers of real-world sensor-node programs.

We have started with an abstract yet intuitive motivation of OSM's main benefits. We have illustrated that event-driven programs have an equivalent OSM representation, which is trivial in the sense that it only consists of a single state. The trivial state machine can then be refined with multiple states and state hierarchies, directly improving its structure and modularity. The resulting state-machine representation can also increase the program's memory efficiency by using state variables, which provide memory reuse without programmer intervention. The reverse transformation cannot be done without loosing those benefits again.

In the second section of this chapter we have taken an experimental approach in showing the positive effects of OSM's automated state management. We have re-implemented a significant part of a large program that was originally implemented in the event-driven NesC programming framework. Then we have compared both approaches. In the example, we have shown that the tedious and error-prone code for state management could be reduced by 70% (from 187 to 56 lines of code), leading to an overall program-size reduction of 31%. A special case of manual state management is extra program code needed to avoid accidental concurrency in event-driven programs. In the example, this extra code accounts for 3% of the entire program. We could show that in OSM typically no countermeasures are required at all.

Finally, we have shown that the state-based approach of OSM has two major benefits over the event-driven code when it comes to initializing resources. The fist benefit is that all initializations of the same state-based resource are co-located (within the scope of an OSM state), rather then being scattered throughout the program code. The second benefit is that equal initializations (e.g., to the same value) need to be written only once. As a result, the code is more concise, more readable and less error prone.

The chapter ends with a section on how memory efficiency can be achieved through state variables. Though quantitative measurements with respect to memory savings are currently still missing, we have illustrated that savings of 10 to 25% are realistic for typical sensor-node programs.

# 8 Related Work

In this chapter we survey approaches and techniques related to sensor-node programming in general, as well as to OSM and its implementation in particular. In the introduction of this dissertation we have described wireless sensor nodes as networked embedded systems. In Section 8.1 we present a brief overview on the state of the art in embedded-systems programming. We also try to answer the question why these approaches have been largely neglected in the wireless sensor-network community. Many of the approaches to embedded-systems programming are based on finite state machines. These approaches are particularly relevant to our work. We will discuss them in Section 8.2 and contrast them with our work. In Chapter 3 we have already presented a number of programming models and frameworks targeted particularly towards sensor nodes. In Section 8.3 we will revisit some of them in order to compare their goals and solution approaches to OSM. Though many of them share the same goals as OSM, the approaches to achieving those goals typically differ significantly.

## 8.1 Programming Embedded Systems

The diversity of embedded systems is reflected in the number and diversity of models and frameworks available for programming them. The majority of them is, however, programmed using conventional methods also found in programming of general-purpose computers. In this section we will first discuss the conventional programming methods for embedded systems before turning to the large number of domain-specific programming frameworks addressing particular requirements of a domain. Sensor nodes are subject to some of the addressed requirements. Therefore such programming frameworks and models are very relevant for wireless sensor-node programming and OSM is based heavily on the concepts found in them. However, besides OSM we are not aware of any other programming frameworks for wireless sensor nodes that draw from such approaches.

### 8.1.1 Conventional Programming Methods

#### Programming Language

The available programming support for a systems crucially depends on its available system resources. So does the supported programming language. The majority of embedded systems today is programmed in the general-purpose programming language C. C is very close to the computational model implemented in today's microcontrollers. Thus there only needs to be a thin adaptation layer resulting in very resource-efficient implementations of C applicable to very resource constrained systems. Often C is used in combination with assembly lan-

guage. A point often made in favor of using assembly language is the full access to the processor's registers and instruction set, thus allowing to optimize for speed. On the other hand, C is considered easy and fast to program as well as more portable between different processors. The portability of C also fosters code reuse and companies often have a large code base which they share among several systems and projects. While C compilers are available for almost all embedded systems, programming languages like C++ and Java are only available for systems with ample resources. The reason is that those languages require a thicker adaptation layer and thus occupy more system resources.

#### System and Programming Support

Today, embedded systems on the high end of the resource spectrum use programming methods very similar to those used for general-purpose computing platforms. Particularly devices such as mobile phones and PDAs have turned into general-purpose computing platforms much more resembling desktop computers than embedded systems. Indeed, some of today's embedded systems posses resources that are similar to those of high-end desktop systems less than a decade ago. At that time, general-purpose operating systems had already pretty much evolved into what they are now. Therefore, it is not surprising that the operating systems found in high-end embedded systems today also support features of state-of-the-art operating systems, including dynamically loadable processes, multi-threading, and dynamic memory management with memory protection. There are several such operating systems available from the open-source domain as well as commercially. The most popular of such systems may be Windows CE, Embedded Linux, PalmOS, VxWorks, and Symbian OS. However, the programming techniques supported by these operating systems are not applicable to the majority of embedded systems and are particularly ill-suited for very resource-constrained sensor nodes.

Systems with resources on the medium to low end of the spectrum typically only have limited system support from a light-weight operating environment or use no operating system at all. Light-weight embedded operating systems are very similar to those found in wireless sensor networks and may have been their archetypes. They typically do not provide system support such as dynamic memory, loadable processes, etc. Many of them, however, provide a multi-threaded execution environment.

As in WSNs, embedded systems at the very low end of the resource spectrum use no operating systems or one providing event-based run-to-completion semantics. With no operating system support, the software is typically implemented as a control loop, which we have already discussed in the context of sensor nodes in Sect. 3.2. As we have discussed extensively in Chapter 4, event-based approaches are not sufficient to program complex yet reliable sensor-node applications with a high degree of concurrency and reactivity.

### 8.1.2 Domain-specific Programming Approaches

Due to their stringent requirements some specific embedded-systems applications and domains require other than the conventional programming meth-

ods. In particular, high demands regarding the reliability of inherently complex and highly concurrent programs call for design methodologies that allow to use high-level design tools, automated testing, simulation, formal verification, and/or code synthesis (i.e., automated code generation). Conventional programming models fail in these domains as they are highly indeterministic and make large-scale concurrency hard to control for the programmer. Additionally, they make reasoning about the memory requirements of a program very hard, particularly if combined with dynamic memory management. As conventional programming approaches do not lend themselves easily to the procedures described above, new models of computation have been suggested and incorporated into design tools. However, the available frameworks and tools supporting such models have been very small in number compared with the more ad hoc designs [77].

Originally, embedded systems have been classified according to their application characteristics into control-oriented systems on the one hand and data-oriented systems on the other hand. However, in recent years there have been several efforts to combine the approaches as embedded systems often expose characteristics of both categories. Today a few powerful tools exist supporting both approaches in a unified model. To explain the principal approaches, we retain the original classification.

### Data-Oriented Systems

Data-oriented systems concentrate on the manipulation of data. They are often called *transformative* as they transform blocks or streams of data. Examples for transformative systems are digital signal processors for image filtering as well as video and audio encoding, which can be found in medical systems, mobile phones, cameras and so on. Since data-processing typically takes a non-negligible time, data-oriented programming frameworks typically need to address real-time requirements.

A common meta model for describing data-oriented systems are dataflow graphs and related descriptions. Such systems are composed of functional units that execute only when input data is available. Data items (also called tokens) are considered atomic units; their flow is represented by directed graphs where each node represents a computational operation and every edge represents a datapath. Nodes connected by a directed edge can communicate by sending data items. Nodes consume the data items received as input and produce new data as output. That is, nodes perform transformational functions on the data items received. Edges may serve as buffers of fixed or (conceptually) unlimited sizes, such as unbounded FIFO queues. Dataflow descriptions are a natural representation of concurrent signal processing applications. There are a number of concrete models based on the dataflow meta model. They differ, for example, in the number of tokens an edge may buffer, whether communication is asynchronous or rendezvous, and whether the number of tokens consumed and produced in an execution is specified a priori.

### Control-Oriented Systems

Instead of computing outputs data from input data, control-oriented embedded systems typically control the device they are embedded into. Often control-oriented systems constantly react to their environment by computing output events from input events (i.e., stimuli from outside of the embedded system). Such systems are called *reactive*. Examples of reactive systems are protocol processors in fax machines as well as mobile phones, control units in automobiles (such as airbag release and cruise control), and fly-by-wire systems.

A subcategory of reactive systems are interactive systems, which also react to stimuli from the environment. However, in interactive systems the computer is the leader of the interaction, stipulating its pace, its allowable input alphabet and order, etc. Therefore, the design of a purely interactive system is generally considered easier than the design of an environmentally-driven reactive system.

Many of the recent applications of embedded systems consist of both, control-oriented and data-oriented parts. In general, the reactive part of the system controls the computationally intense transformative operations. In turn, results of the such operations are often feed back into the control-oriented part. Thus, actual embedded systems often expose both characteristics.

### Control and Data-oriented Wireless Sensor Networks

Wireless sensor nodes, too, include transformational as well as reactive parts. For example, transformational parts in sensor nodes compute from streams of data whether a product has been damaged [106] (acceleration data), the location where a gun has been fired [86] (audio data), the specific type of animal based on its call [64, 105, 117, 118] (again, audio data), and so on. The same sensor nodes also have control-oriented parts, for example, for processing network-level and application-level protocols (such as the Bluetooth protocol stack used in [106] and the EnviroTrack group-management application protocol discussed in the previous chapter). In WSNs, the feedback loop between the reactive and transformative parts is of particular importance. Due to the constrained nature of the nodes, resource-intense operations are often controlled by criteria that are themselves computed by a much less intense processing stage. In [118], for example, recorded audio samples are first rated how well they match calls of specific birds using a relatively simple algorithm. Only if there is a high confidence that a sample represents a desired birdcall, the samples are compressed and sent to the clusterhead in order to apply beamforming for target localization, which are very resource-intense operations. Other criteria for controlling the behavior of sensor nodes may reflect the importance of a detection, the proximity (and thus expected sensing quality) of a target, the number of other nodes detecting the same target, and so on. In the extreme case, so called adaptive-fidelity algorithms allow to dynamically trade the quality of a sensing result against resource usage by controlling the amount of sampling and processing performed by a node.

### 8.1.3 Embedded-Systems Programming and Sensor Nodes

The work by the embedded systems community on techniques such as real-time systems, model driven design, hardware-software co-design, formal verification, code synthesis, and others share many goals with WSN research and has led to a variety of novel programming models and frameworks. Despite their apparent relevance, alternatives to these approaches from the embedded systems community seem to have attracted little attention in the wireless sensor network community. The most popular programming frameworks for WSNs adopt the prevalent programming paradigm: a sequential computational model combined with a thread-based or event-based concurrency model.

We have no comprehensive answer why this is so. We can only speculate that the sequential execution model inherent to the most popular general-purpose programming languages (such as C, C++, Java) has so thoroughly pervaded the computer-science culture that we tend to ignore other approaches (cf. [77]). Despite the potential alternatives, even the majority of traditional embedded systems is still programmed using the classical embedded programming language C, and recently also C++ and Java. One reason may be that there seems to be little choice in the commercial market. The most popular embedded operating systems fall in this domain. Another reason may be that the vast majority of programmers has at least some experiences in those languages whereas domain-specific embedded frameworks are mastered only by a selected few. Consequently, many embedded software courses toughed at universities also focus on those systems (see, for example, [92]). In the next section we will discuss concrete programming models for embedded systems that are most relevant to OSM.

## 8.2 State-Based Models of Computation

OSM draws directly from the concepts found in specification techniques for control-oriented embedded systems, such as finite state machines, Statecharts [53], its synchronous descendant SyncCharts [10, 11], and other extensions. From Statecharts, OSM borrows the concept of hierarchical and parallel composition of state machines as well as the concept of broadcast communication of events within the state machine. From SyncCharts we adopted the concept of concurrent events.

Variables are typically not a fundamental entity in control-oriented state-machine models. Rather, these models rely entirely on their host languages for handling data. Models that focus both on the transformative domain and the control-oriented domain typically include variables. However, we are not aware of that any of these models uses machines states as a qualifier for the scope and lifetime of variables.

### 8.2.1 Statecharts

The Statecharts model lacks state variables in general and thus does not define an execution context for actions in terms of data state. Also, the association of entry and exit actions with states *and* transitions, that is, distinct incoming

and outgoing actions, is a distinctive feature of OSM. These associations clearly define the execution context of each computational action to be exactly one program state and the set of state variables in the scope of that state. While the Statecharts model, too, has entry and exit actions, they are only associated with a state and are invoked whenever that state is entered or left, respectively.

### 8.2.2 UML Statecharts

The Unified Modeling Language (UML) [49] is a graphical language for specifying, visualizing, and documenting various aspects of object-oriented software. UML is composed of several behavioral models as well as notation guides and UML Statecharts is one of them. UML Statecharts are a combination of the original Statecharts model and some object-oriented additions. They are intended for modeling and specifying the discrete behavior of classes. UML is standardized and maintained by the Object Management Group (OMG). A number of additions to UML Statecharts have been proposed for various application domains. UML Statecharts are supported by several commercial tools, for example, VisualSTATE [121]. VisualSTATE has introduced typed variables as an extension to UML Statecharts, however, all variables are effectively global. Another commercial tool supporting UML Statecharts is Borland Together.

### 8.2.3 Finite State Machines with Datapath (FSMD)

Finite State Machines with Datapath (FSMD) [44] are similar to FSM (neither hierarchical nor parallel). Variables have been introduced in order to reduce the number of states that have to be declared explicitly. Like OSM, this model allows programmers to choose to specify program state explicitly (with machine states) or implicitly with variables. As in OSM, transitions in FSMD may also depend on a set of internal variables. FSMD are flat, that is, they do not support hierarchy and concurrency, and variables have global scope and lifetime. On the Contrary, variables in OSM are bound to a state hierarchy. The FSMD model is used for hardware synthesis (generating hardware from a functional description).

### 8.2.4 Program State Machines (PSM), SpecCharts

The Program-State Machine (PSM) meta model [114] is a combination of hierarchical and concurrent FSMs and the sequential programming language model where leave states may be described as an arbitrarily complex program. Instantiations of the model are SpecCharts [114] and SpecC [43].

In PSM a program state $P$ basically consists of program declarations and a *behavior*. Program declarations are variables and procedures, whose scope is $P$ and any descendants. A behavior is either a sequence of program statements (i.e., a regular program) without any substates, or a set of program concurrently executing substates (each having its own behavior), or a set of sequentially-composed substates and a set of transitions. Transitions have a $type$ and a $condition$. Transitions are triggered by a condition represented by a boolean expression. The transition type is either transition-on-completion (TOC) or

transition-immediately (TI). TOC transitions are taken if and only if the source program state has finished its computation and condition evaluates to true. TI transitions are taken when the condition evaluates to true, that is, they terminate the source state immediately, regardless of whether the source state has finished its computations. TI transitions can be thought of as exceptions.

Similar to OSM, variables are declared within states; the scope of a variable then is the state it has been declared in and any descendants. The lifetime of variables, however, is not defined. The main difference to OSM is, that computations in SpecCharts are not attached to transitions but rather to leaf states (i.e., uncomposed states) only. In analogy to Moore and Mealy machines, we believe that reactive systems can be specified more concisely in OSM. Though both models are computationally equivalent, converting a Mealy machine (where output functions are associated with transitions) to a Moore machine (output associated with states) generally increases the size of the machine, that is, the number of states and transitions. The reverse process leads to fewer states. Finally, in contrast to PSMs, OSM allows to access the values of events in computational actions. A valued event is visible in the scope of both the source and the target state of a transition (in "out" and "in" actions, respectively). This is an important aspect of OSM.

SpecCharts [114] are based on the Program-State Machine (PSM) meta model using VHDL as programming language. It can be considered as a textual state-machine extension to VHDL. SpecCharts programs are translated into plain VHDL using an algorithm described in [93]. The resulting programs can then be subjected to simulation, verification, and hardware synthesis using VHDL tools.

### 8.2.5 Communicating FSMs: CRSM and CFSM

A model for the design of mixed control and data-oriented embedded systems are *communicating FSMs*, which conceptually separate data and control flow. In this model, a system is specified as a finite set of FSMs and data channels between pairs of machines. FSM execute independently and concurrently but communicate over typed channels. Variables are local to a single machine, but global to the states of that machine. Values communicated can be assigned to variables of the receiving machine. There are several variations of that basic model. For example, in Communicating Real-Time State Machines (CRSM) [104] communication is synchronous and unidirectional. Individual FSMs are flat. Co-design Finite State Machines (CFSM) [13] communicate asynchronously via single element buffers, but FSM may be composed hierarchically. In contrast to communicating FSMs, concurrent state machines in OSM communicate through events or shared variables.

### 8.2.6 Esterel

OSM, like SyncCharts, is implemented on top of Esterel [26]. We considered using Esterel directly for the specification of control flow in OSM. However, as an imperative language, Esterel does not support the semantics of FSM directly. We believe that FSM are a very natural and powerful means to model WSN

applications. Moreover, specifications in Esterel are generally larger (up to 5 times) compared to OSM.

### 8.2.7 Functions driven by state machines (FunState)

FunState [110] (functions driven by state machines) is a design meta model for mixed control and data flow. FunState was designed to be used as an internal representation model used for verification and scheduling and has no specific host language.

FunState unifies several well-known models of computation, such as communicating FSMs, dataflow graphs, and their various incarnations. The model is partitioned in a purely reactive part (state machine) without computations and a passive functional part (a "network" resembling PetriNets). Transitions in the state-machine part trigger functions in the network. The network consists of a set of storage units, a set of functions and a set of directed edges. Edges connect functions with storage units and storage units with functions. Edges represent the flow of valued data tokens.

FunState has a synchronous model of time where in an atomic step tokens are removed from storage units, computations are executed, and new tokens are added to storage units. Real time properties of a system can be described with timed transitions and time constraints. Functions have run-to-completion semantics; they are executed in non-determinate order.

## 8.3 Sensor-Node Programming Frameworks

We have presented the state-of-the-art system software in Chapter. 3. In this section we will revisit some of them focusing on systems that share the same goals as OSM or use a programming model related to state machines.

### 8.3.1 TinyOS

The design of TinyOS shares two main common goals with OSM, namely to provide efficient modularity as well as to provide a light-weight execution environment, particularly in terms of memory consumption.

#### Modularity

Just as OSM, TinyOS has been designed with a high degree of modularity in mind. The designers argue that sensor-node hardware will tend to be application specific rather than general purpose because of the wide range of potential applications. Thus, on a particular device the software should be easily synthesizable from a number of existing and custom components. TinyOS addresses modularity by the concept of components. NesC components expose a well-defined interface, encapsulating a self-confined piece of software (such a timer and access to a particular sensor). Each of the pieces can be implemented using the event-driven model of NesC.

NesC components foster program modularity, as components with different implementations yet a common interface can be exchanged easily. They also

provide a means to structure complex programs or components into a number of less-complex components. NesC components partition a program in logically disjunct pieces, each of which is more easily manageable as the entire program. Each component is active during the entire runtime of the program. NesC components are a major improvement over basic event-driven systems (such as the BTnode system software [21]) that only provide the concept of actions to structure a program.

To address modularity, OSM relies on the concept of program states. OSM programs can be composed from a number of less-complex states. The composition is either strictly hierarchical or concurrent. Unlike NesC components, states in OSM do not only partition the program logically but also in the time domain. Since programs typically exhibit time-dependent behavior, OSM provides more flexibility in modeling a program or logical component thus resulting in more-structured program code.

The implementation of NesC components has one major drawback. NesC components define the scope for variables declared within them, that is, variables of a particular component cannot be accessed from other components. As a consequence, the communication among NesC components has to rely on the passing of (one of the two types of) events with arguments. For performance reasons and to limit the number of events passed, programmers often duplicate variables in several modules. For example, in the EnviroTrack middleware, group-management messages are sent from a number of components whereas the functionality to actually send the message is performed in a dedicated component. This component can process a single message at a time (i.e., there are no message buffers) and has a boolean variable indicating whether the previous message has already been sent completely. However, in order to avoid the overhead of composing a group-management message while the network stack is not ready to send the next message, the boolean variable is replicated in four modules. This drawback has been one of the reasons for the development of TinyGALS [29].

#### Light-weight Execution Environment

A primary goal that TinyOS shares with OSM is the provision of a very light-weight execution environment. In order to handle concurrency in a small amount of stack space and to avoid per-thread stacks of multi-threaded systems, the event-driven model has been chosen. Similar to our approach, NesC provides asynchronously scheduled actions as a basic programming abstraction. As in OSM, events are stored in a queue and processed in first-in-first-out order by invoking the associated action. However, NesC also provides programmers access to interrupt service routines (ISRs) so that a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted. Such hardware events can be propagated to a stack of NesC components. In contrast, interrupts in OSM are hidden from the programmer. The reason for these differences are contrasting assumptions on the separation of application and execution environment: while in TinyOS programmers are expected to program their own low-level drivers, for example, for sensors and radio transceivers, OSM is based on the assumption that such

low-level interfaces are provided by the node's operating system.

While in OSM actions are never interrupted, ISRs in the NesC can interrupt actions as well as other ISRs. Actions run to completion only with respect to other actions. Therefore, a NesC application may be interrupted anywhere in an action or ISR. This model can lead to subtle race conditions, which are typical for multi-threaded systems but are not found in typical atomic event-based systems. To protect against race conditions, programmers of TinyOS need to enforce atomic execution of critical sections using the `atomic` statement.

Though TinyOS does not require per-thread stacks of regular multi-threading systems, they share some of the undesirable characteristics. As discussed in Sect. 3.2.4, interruptible (i.e., non-atomic) code is hard to understand and debug and therefore potentially less reliable. Also, there a may be issues with modularity.

### 8.3.2 Contiki

Contiki [37] has been designed as a light-weight sensor-node operating system. In order to work efficiently within the constrained resources, the Contiki operating system is built around an event-driven kernel. Contiki is particularly interesting as it also supports preemptive multi-threading through loadable modules. In Contiki, a single program can be built using a combination of the event-driven model as well as the multi-threaded model. The authors suggest to build applications only based on events whenever possible. However, individual, long-running operations, such as cryptographic operations, can be specified in separate threads. OSM on the contrary does not support multiple threads and only allows to specify actions of bounded time.

Building OSM on the Contiki operating system would open up the possibility to implement *activities* as proposed by [53]. Activities can be viewed as actions that are created on the occurrence of an event and which may run concurrently with the rest of the state machine. They terminate by emitting an event, which may carry a result as parameter.

### 8.3.3 Protothreads

Protothreads [38] attack the stack-memory issues of multi-threading from another perspective. Protothreads provide a multi-threaded model of computation that do not require individual stacks. Therefore, Protothreads eliminate what is probably the dominant reason why sensor-node operating systems are based on the event-based model. This comes, however, at the cost of local variables. That is, stackless Protothreads do not support automatically memory-managed local variables that are commonplace in modern procedural languages. Removing the per-thread stack also effectively removes the main mechanism to save memory on temporary variables.

As a consequence, the same issues as in event-driven programming arise: either data state has to be stored in global variables, which is wasteful on the memory and has to be manually memory managed. Or dynamic memory management is required, which has the drawbacks explained in Sect. 3.3 and still requires significant manual intervention from a programmer. Effectively, the

memory issues have been shifted from the operating system to the application program, which now has to take care of managing temporary variables. As a consequence, applications written with Protothreads may require more memory compared to regular threads, leading to a negative grand total.

### 8.3.4 SenOS

Like OSM, SenOS [75] is a state-based operating system for sensor nodes. It focuses on dynamic reconfigurability of applications programs. Unlike OSM, programs are specified as flat and sequential state machines, each of which is represented by a separate state-transition table. As in OSM, actions are associated with state transitions. All available actions are stored in a static library that is transferred to the sensor node at the time of programming together with the operating system kernel. Applications are implemented as state-transition tables. These tables define all possible machine states and transitions. They also associate actions from the action library to particular transitions.

SenOS state machines can run concurrently, where each machine runs in its own thread. Applications programs (as specified by state transition tables) can be dynamically added to and removed from the system at runtime. However, the static library of actions cannot be changed unless reprogramming the entire sensor node.

SenOS relies on task switches for concurrency. Its concurrency permits dynamic scheduling of tasks. In contrast, OSM features a static concurrency model. The static concurrency model allows very memory efficient implementations without the need to provide a separate runtime stack for each concurrent task. Furthermore, the static concurrency model makes OSM programs amenable to formal verification techniques found in state-machine approaches. The flat state machines approach precludes the use of automatic scoping and lifetime mechanisms for variables, which is a major design point in OSM. It is questionable if the desired re-configurability and re-programmability can be achieved with this approach, since programs have to be built exclusively on the preconfigured actions library without any glue code.

# 9 Conclusions and Future Work

In this final chapter, we conclude by summarizing the contributions of our work and by discussing its limitations. We also propose future work addressing concrete limitations of our approach.

## 9.1 Conclusions

Event-driven programming is a popular paradigm in the domain of sensor networks in general. For sensor networks that are operating at the very low end of the resource spectrum it is in fact the predominant programming model. Unlike the multi-threaded programming model, system support for event-driven programming requires very little of a system's resources. The event-driven programming model has been adopted by a large number of programming frameworks for sensor networks, among them TinyOS / NesC, which currently may be the most popular of all.

Despite its popularity, the event-driven programming model has significant shortcomings. Particularly in large and complex programs these shortcomings lead to issues with the readability and structure of the program code, its modularity and correctness, and, ironically, also the memory efficiency of the developed programs. Concretely, an event-driven program typically uses more RAM as a functionally equivalent sequential program because in the event-driven model a lot of temporary data has to be stored in global variables. A sequential program would use local variables instead, which are automatically memory managed and thus their memory is reused.

With respect to these problems, the main contribution of this dissertation is to show how the event-based model can be extended so that it allows to specify well-structured, modular, and memory-efficient programs, yet requires as little runtime support as the original model. We have significantly improved sensor-node programming by extending the event-driven model to a state-based model with a explicit notion of hierarchical and concurrent program states. We have also introduced a novel technique to use states as a scoping and lifetime qualifier for variables, so they can be automatically memory managed. This can lead to significant memory savings in temporary data structures. In the following we will list our contributions towards our solution in more detail.

## 9.2 Contributions

Below we will summarize our contributions towards the problem analysis, the solution approach and its implementation, as well as the evaluation. Some contributions have also been published in [21, 72, 74].

### 9.2.1 Problem Analysis: Shortcomings of Event-driven Programming

In Chapter 4 we have contributed a thorough analysis of the original event-driven model in the context of sensor networks. Firstly, we have analyzed the two main problems of the existing event-driven programming paradigm. While the basic problem leading to unstructured and un-modular program code, namely manual state management, has already been identified and documented in previous work, the memory inefficiency of event-driven sensor-node programs incurred by manual state management has not been analyzed before.

Secondly we have contributed by analyzing the anatomy of sensor node applications. We have found that, though easy and intuitive for small and simple programs, event-driven programs do not describe typical sensor-node applications very well. That is, the conceptual models that programmers create in their minds before implementing a particular program or algorithm does not map easily to the event-driven programming model. In particular, we have identified the lack of an abstraction that allows programmers to structure their programs along the time domain into discrete program phases. We have shown that several algorithms from the literature are indeed described as systems partitioned into discrete phases, each having distinctive data state and behavior. The findings of our problem analysis now allows to better evaluate current and future programming models for sensor networks.

### 9.2.2 Solution Approach: State-based Programming with OSM

To alleviate the problems described above, we have proposed a sensor-node programming model that is extending the event-driven model with well-known state-machine abstractions. In Chapter 5 we contribute by presenting such a model, which we call the OBJECT STATE MODEL (OSM). OSM is based on abstractions of hierarchically and concurrent state machines. Though they have been used successfully in embedded-systems programming for several years, they have not yet been applied to the field of sensor networks.

Besides relying on proven programing abstractions, our OSM model also introduces the concept of *state variables*. State variables supports the memory-efficient storage and use of temporary data. State variables resemble local variables of sequential programs; both are automatically memory managed and thus their memory is automatically reused when no longer needed. State variables can help overcome the inherent memory inefficiency of event-driven programming for storing temporary data. Besides that, the maximum memory requirements of all parts of even highly-concurrent programs can be calculated at compile time. This allows to check whether a particular program can run on a given sensor node and to easily identify program parts susceptible for optimizations.

Finally, OSM introduces the concept of *entry and exit actions* (also referred to as incoming and outgoing actions) to traditional state-machine programming models. They allow to elegantly and efficiently specify the initialization and release of state-specific resources, which are common tasks in real-world sensor-node programs.

### 9.2.3 Prototypical Implementation

We have implemented a prototypical OSM compiler which generates C-language code, which we have presented in Chapter 6. The compiler translates state-based OSM programs into event-driven program representations. OSM programs only require the same minimal runtime support as programs initially written with event-driven frameworks. Particularly, we show that the code generated from OSM specifications runs on a only slightly modified version of our light-weight and event-driven system software for the BTnode sensor node. Though the compilation introduces some memory overhead into the binary, this overhead can be attributed to the inefficiencies introduced by using an additional intermediate language (namely Esterel) in the prototypical compiler. The prototypical implementation of an OSM compiler shows that compiling OSM programs to efficient event-driven programs is generally feasible.

### 9.2.4 Evaluation

In Chapter 7 we have shown how automating state and stack management benefits programmers of real-world sensor-node programs. We have shown the practical feasibility by presenting an OSM-based re-implementation of part of EnviroTrack, a system for tracking mobile objects with sensor networks, and other algorithms.

We have re-implemented a significant part of the EnviroTrack sensor-network middleware, which was originally implemented in the event-driven NesC programming framework. Using that re-implementation we demonstrate the usefulness of state-based representations. We have shown that the tedious and error-prone code for state management could be reduced significantly. In the example, 187 lines of manual state-management code were reduced to an explicit state representation of only 56 lines, representing a reduction of 70%. The overall program-size reduction was 31%. An additional 16 lines of the original EnviroTrack code are used to guard against the unwanted execution of actions in certain contexts, which is also a form of manual state-management. In OSM, no extra code is required because the model explicitly specifies which actions can be called in which context. Finally, we have shown that the OSM code is more structured. We have shown that logically-related code in OSM is also co-located in the implementation whereas it is typically scattered throughout much of the implementation in event-driven code. For example, in OSM there is typically a single place for initializing particular timers, which is also adjacent to the actual action triggered by the timer. In NesC, however, there are several isolated places in the code where the timer is initialized, which are also disconnected from the triggered action.

## 9.3 Limitations and Future Work

There are a number of limitations and potential improvements with respect to our approach and methodology, which we will discuss in this section. We also discuss future work in the broader context of our work.

### 9.3.1 Language and Program Representation

Although the we have shown that the state-based program representation of OMS can lead to more modular and structured code, there is still much to improve.

The OSM compiler is currently only implemented for our textual input language, though we have also sketched a possible graphical representation throughout this dissertation. We found that a graphical representation is better suited to present the big picture of a program (i.e., its coarse-grained structure), particularly for reviewing the design, for which the textual representation is less intuitive. Therefore we have generally provided both program representations, the graphical as well as the textual, in examples of this dissertation. On the other hand, the textual notation is very efficient for specifying implementation detail, such as action names, parameter names, etc. In our experience with graphical tools for UML Statecharts (namely VisualSTATE and Borland Together, both commercial tools) and for SyncCharts (the freely available `syncCharts` editor [76]) the specification of implementation details was unsatisfactory. Therefore it would be beneficial to support both representations, allowing to freely switch between and to use the one which is more suitable for the task at hand.

#### New Model

As in the event-driven programming model, actions in OSM must be non-blocking. We have found that programmers who had used only sequential programming languages before, have difficulties in understanding this. For example, students programming with our event-driven BTnode system software regularly used `while`-loops in one action polling a variable that is set in another action, wondering why the program would block forever. In general, OSM requires most programmers to get used to new concepts and to the OSM specification language in particular. Though this is not a limitation of the programming model as such, it may hamper its widespread use. OSM does provide semantics that are compatible with existing event-based systems, however, thus easing the transition for programmers that are already familiar with event-based programming.

#### Representing Sequential Program Parts

Purely sequential parts of the program flow (i.e., linear sequences of events and actions without any branches) are tedious to program in OSM because they have to be explicitly modeled as sequences of states, pairwise connected by a single transition (see the `BUSY` state of Prog. 7.4 on page 133 as an example). In such cases wait-operations, as typically provided by multi-threaded programming models, might be more natural. To solve this issue, it may be worth investigating how Protothreads [38] could be used to implement sequential program parts.

#### Representing Composite Events

Likewise, reactions to composite events (i.e., meta-events made up of multiple events, for example, "events $e1$, $e2$, and $e3$ in any order") cannot be specified

concisely. Instead, all intermediate states and transitions have to be modeled explicitly even though only the reaction to the event completing the composite event may be relevant. Introducing more elaborate triggers of transitions and a concise notation could solve this problem. Such a trigger mechanism could be implemented as semantic sugar using the existing concepts of states and transitions but hiding them from the user.

### 9.3.2 Real-Time Aspects of OSM

OSM does not address real-time behavior. It does not provide any mechanisms to specify deadlines and to determine how much time transitions and their associated actions take. There is, however, a large body of work on state-machine based real-time systems. Future work could investigate how these approaches can be integrated into OSM.

Also, if actions generate events, the order of events in the queue (and hence the system behavior) may depend on the execution speed of actions.

### 9.3.3 Memory Issues

#### Queue Size and Runtime-stack Size Estimations

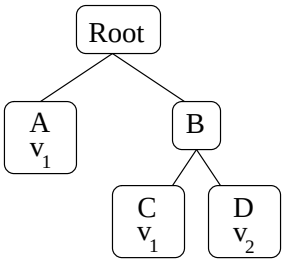
One of the reasons why OSM has been developed is to resolve the memory issues of temporary-data management, which is prevalent in event-driven programming. While OSM can calculate the program's memory usage at compile time, the maximum queue-size (required for storing events that cannot be processed immediately) remains unknown. As a consequence, the event queue may be oversized, thus wasting memory, or undersized, resulting in lost data at best but more likely in a program crash. Future work could estimate the maximum queue size required.

The other unknown memory factor besides the queue size is the system's runtime stack-size. The runtime stack is used for holding information of function calls (i.e., its parameters and the return address) as well as the local variables of OSM actions and regular functions of the host language. (State variables are stored statically and do not use the stack.) Being able to precisely estimate the stack size would be beneficial for the reasons described above. There has been some work on estimating stack sizes, particularly in the context of multithreaded programs. In future work the OSM compiler could be extended to perform such an estimate.

#### Applicability of State Variables

State variables can be tied to entire state hierarchies only. However, a more flexible scoping mechanism of state variables may be desirable. Consider the state hierarchy depicted below where a variable $v_1$ is being used in the states $A$ and $C$ but not in $D$. In OSM such a variable would be modeled as a state variable of root $R$. Its scope and lifetime would extend to all substates of $R$ including $D$. In the given case, OSM is not able to reuse the memory holding $v_1$ in $D$, for example, for $D$'s own private variable $v_2$. Flexible scoping mechanism that allows to include selected states only (as opposed to being bound to the state

hierarchy) could further increase the memory efficiency but are left for future work.




## 9.4 Concluding Remarks

There are still features and tools missing to make OSM an easy and intuitive to use programming framework for resource constrained sensor nodes. Particularly more experience is required to evaluate the value of OSM for large projects and to determine what needs to be included in future versions. However, we believe that OSM is a big step towards extending the application domain of event-based programming to larger and more complex systems.

# A OSM Code Generation

This Appendix demonstrates the stages of the code generation process employed by our OSM compiler prototype for a small, yet complex OSM program example. In Sect. A.1 we present this program, both in graphical as well as in textual representation. Sect. A.2 presents the C-language include file that is generated from the OSM example program. It includes the mapping of the OSM program's variables to a C-language struct as well as the prototypes of all actions defined in the OSM program. Sect. A.3 present the control flow mapping of the OSM program to the Esterel language. The generated Esterel code is translated to the C language by an Esterel compiler. The compiler's output is presented in Sect. A.4. A functionally equivalent version optimized for memory efficiency is also presented. Finally in Sections A.5.1 and A.5.2 we present the Makefile for compiling the OSM example program and the output of an OSM compiler during a compilation run.

## A.1 OSM Example

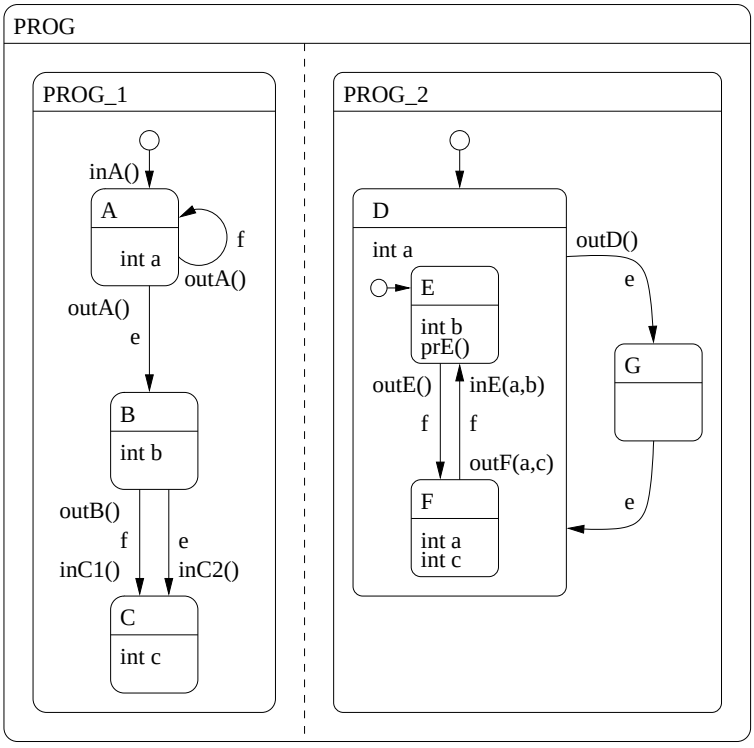



Figure A.1: Graphical representation of `test.osm`, a program for testing the OSM compiler.

```
// -------------------------------------------------------
// test.osm -- A program for testing the OSM compiler
// -------------------------------------------------------

state PROG {
  state PROG_1 {

    initial state A {
      var int a;
      onEntry: start/ inA();
      e / outA() -> B;
      f / outA() -> A;
    } // end state A

    state B {
      var int b;
      f / outB() -> C;
      e /        -> C;
    } // end state B

    state C {
      var int c;
      onEntry: B -> f/ inC1();
      onEntry: B -> e/ inC2();
    } // end state C

  } // end PROG_1
||
  state PROG_2 {
    initial state D {
      var int a;

      initial state E {
        var int b;
        onEntry: f / inE(a, b);
        f / outE() -> F;
        // onPreemption: prE();
      } // end state E
      state F {
        var int a;
        var int c;
        f / outF(a, c) -> E;
      } // end state F

      e / outD() -> G;
    } // end state D

    state G { e / -> D; }
  } // end PROG_2
} // end PROG
```

## A.2 Variable Mapping (C include file)

```
/* This include file was generated by OSM on
 * Samstag, Juni 24, 2006 at 09:31:50.
 * OSM compiler written by Oliver Kasten <oliver.kasten(at)inf.ethz.ch> */


/* prevent the esterel compiler from declaring functions extern */
#define _NO_EXTERN_DEFINITIONS

// function for making a discrete step in the state automaton
int STATE_PROG();
// function for resetting all inputs of the state automaton
int STATE_PROG_reset();

/* no outputs events from state PROG */

/* exit actions prototypes */
void outA();
void outB();
void outD();
void outE();
void outF(int*, int*);
#define outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c() outF( \
  &_statePROG._par_PROG_1_PROG_2._statePROG_2._seq_D_G._stateD._seq_E_F._stateF.a, \
  &_statePROG._par_PROG_1_PROG_2._statePROG_2._seq_D_G._stateD._seq_E_F._stateF.c )

/* entry actions prototypes */
void inA();
void inC1();
void inC2();
void inE( int*, int*);
#define inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b() inE( \
  &_statePROG._par_PROG_1_PROG_2._statePROG_2._seq_D_G._stateD.a, \
  &_statePROG._par_PROG_1_PROG_2._statePROG_2._seq_D_G._stateD._seq_E_F._stateE.b )

/* preemption actions */
void prE();

struct statePROG {
  /* parallel machine */
  union par_PROG_1_PROG_2 {
   /* contained states */
    struct statePROG_1 {
      /* sequential machine */
      union seq_A_B_C {
       /* contained states */
        struct stateA {
          int a;
        } _stateA;
        struct stateB {
          int b;
        } _stateB;
        struct stateC {
          int a;
        } _stateC;
      } _seq_A_B_C;
    } _statePROG_1;
    struct statePROG_2 {
      /* sequential machine */
      union seq_D_G {
       /* contained states */
        struct stateD {
          int a;
          /* sequential machine */
          union seq_E_F {
           /* contained states */
            struct stateE {
              int b;
            } _stateE;
            struct stateF {
              int a;
              int c;
```

```
72              } _stateF;
73            } _seq_E_F;
74          } _stateD;
75          struct stateG {
76          } _stateG;
77        } _seq_D_G;
78      } _statePROG_2;
79    } _par_PROG_1_PROG_2;
80  } _statePROG;
```

# A.3 Control Flow Mapping–Stage One (Esterel)

```
% This Esterel code was generated by OSM on
% Samstag, Juni 24, 2006 at 06:41:44.
% OSM compiler written by Oliver Kasten <oliver.kasten(at)inf.ethz.ch>

% o-star
module STATE_PROG:
  input e, f;
  % no outputs
  % no function declarations
    run MACRO_PROG_1_PROG_2; % parallel substates
    halt
end module % STATE_PROG

module MACRO_PROG_1_PROG_2:
  input e, f;
  % no output
    trap T in
      run STATE_PROG_1
      ||
      run STATE_PROG_2
    end trap
end module % MACRO_PROG_1_PROG_2

% o-star
module STATE_PROG_1:
  input e, f;
  % no outputs
  % no function declarations
    signal start_PROG_1 in
      emit start_PROG_1; % emit pseudo event start for superstate PROG_1
      run CONSTELLATION_A_B_C; % sequential substates
    end signal;
    halt
end module % STATE_PROG_1

module CONSTELLATION_A_B_C:
  input e, f, start_PROG_1;
  % no output

  signal gamma1, gamma2, gamma3 in
    % init:
    emit gamma1; % start initial state
    % mainloop
    loop
      trap L in
        % STATE_A
        present gamma1 then
          signal alpha in
            emit alpha;
            run STATE_A [signal gamma1/ xi1, gamma2/ xi2];
            present alpha else exit L end present
          end signal
        end present;
        % STATE_B
        present gamma2 then
          signal alpha in
            emit alpha;
            run STATE_B [signal gamma3/ xi1, gamma3/ xi2];
            present alpha else exit L end present
          end signal
        end present;
        % STATE_C
        present gamma3 then
          signal alpha in
            emit alpha;
            run STATE_C;
            present alpha else exit L end present
          end signal
        end present;
      halt % never reached
      end trap
    end loop
  end signal
end module % CONSTELLATION_A_B_C

module STATE_A:
  input e, f, start_PROG_1;
  output xi1, xi2;
  procedure inA()(), outA()();
  % onEntry actions
  present
    case start_PROG_1 do call inA()();
  end present;
```

```
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case f do emit iota; emit xi1; call outA()(); exit T
          case e do emit iota; emit xi2; call outA()(); exit T
        end present
      end every
    end signal
  end trap
end module % STATE_A

module STATE_B:
  input e, f;
  output xi1, xi2;
  procedure outB()();
  % no onEntry actions
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case f do emit iota; emit xi1; call outB()(); exit T
          case e do emit iota; emit xi2; exit T
        end present
      end every
    end signal
  end trap
end module % STATE_B

% o-star
module STATE_C:
  input e, f;
  % no outputs
  procedure inC1()(), inC2()();
  present
    case f do call inC1()();
    case e do call inC2()();
  end present;
    halt
end module % STATE_C

% o-star
module STATE_PROG_2:
  input e, f;
  % no outputs
  % no function declarations
    run CONSTELLATION_D_G; % sequential substates
    halt
end module % STATE_PROG_2

module CONSTELLATION_D_G:
  input e, f;
  % no output

  signal gamma1, gamma2 in
    % init:
    emit gamma1; % start initial state
    % mainloop
    loop
      trap L in
        % STATE_D
        present gamma1 then
          signal alpha in
            emit alpha;
            run STATE_D [signal gamma2/ xi1];
            present alpha else exit L end present
          end signal
        end present;
        % STATE_G
        present gamma2 then
          signal alpha in
            emit alpha;
            run STATE_G [signal gamma1/ xi1];
            present alpha else exit L end present
```

```
          end signal
        end present;
      halt % never reached
      end trap
    end loop
  end signal
end module % CONSTELLATION_D_G

module STATE_D:
  input e, f;
  output xi1;
  procedure prE()(), outD()();
  % no onEntry actions
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          run CONSTELLATION_E_F; % sequential substates
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case e do emit iota; emit xi1; call prE()(); call outD()(); exit T
        end present
      end every
    end signal
  end trap
end module % STATE_D

module CONSTELLATION_E_F:
  input f;
  % no output

  signal gamma1, gamma2 in
    % init:
    emit gamma1; % start initial state
    % mainloop
    loop
      trap L in
        % STATE_E
        present gamma1 then
          signal alpha in
            emit alpha;
            run STATE_E [signal gamma2/ xi1];
            present alpha else exit L end present
          end signal
        end present;
        % STATE_F
        present gamma2 then
          signal alpha in
            emit alpha;
            run STATE_F [signal gamma1/ xi1];
            present alpha else exit L end present
          end signal
        end present;
      halt % never reached
      end trap
    end loop
  end signal
end module % CONSTELLATION_E_F

module STATE_E:
  input f;
  output xi1;
  procedure inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b()(), outE()();
  % onEntry actions
  present
    case f do call inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b()();
  end present;
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case f do emit iota; emit xi1; call outE()(); exit T
        end present
      end every
    end signal
  end trap
```

```
end module % STATE_E

module STATE_F:
  input f;
  output xi1;
  procedure outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c()();
  % no onEntry actions
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case f do emit iota; emit xi1;
                    call outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c()(); exit T
        end present
      end every
    end signal
  end trap
end module % STATE_F

module STATE_G:
  input e;
  output xi1;
  % no function declarations
  % no onEntry actions
  trap T in
    signal alpha, iota, omega in
      emit alpha;
      suspend
        % suspend
          halt
        % when <g0>
      when immediate iota
      ||
      every tick do
        present
          case e do emit iota; emit xi1; exit T
        end present
      end every
    end signal
  end trap
end module % STATE_G
```

# A.4 Control Flow Mapping–Stage Two (C)

## A.4.1 Output of the Esterel Compiler

```
/* sscc : C CODE OF SORTED EQUATIONS STATE_PROG - INLINE MODE */

/* AUXILIARY DECLARATIONS */

#ifndef STRLEN
#define STRLEN 81
#endif
#define _COND(A,B,C) ((A)?(B):(C))
#ifdef TRACE_ACTION
#include <stdio.h>
#endif
#ifndef NULL
#define NULL ((char*)0)
#endif
```

[...]

```
#include "test.h"

/* EXTERN DECLARATIONS */

#ifndef _NO_EXTERN_DEFINITIONS
#ifndef _NO_PROCEDURE_DEFINITIONS
#ifndef _inA_DEFINED
#ifndef inA
extern void inA();
#endif
#endif
#ifndef _outA_DEFINED
#ifndef outA
extern void outA();
#endif
```

```
#endif
#ifndef _outB_DEFINED
#ifndef outB
extern void outB();
#endif
#endif
#ifndef _inC1_DEFINED
#ifndef inC1
extern void inC1();
#endif
#endif
#ifndef _inC2_DEFINED
#ifndef inC2
extern void inC2();
#endif
#endif
#ifndef _prE_DEFINED
#ifndef prE
extern void prE();
#endif
#endif
#ifndef _outD_DEFINED
#ifndef outD
extern void outD();
#endif
#endif
#ifndef _inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b_DEFINED
#ifndef inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b
extern void inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b();
#endif
#endif
#ifndef _outE_DEFINED
#ifndef outE
extern void outE();
#endif
#endif
#ifndef _outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c_DEFINED
#ifndef outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c
extern void outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c();
#endif
#endif
#endif
#endif

/* MEMORY ALLOCATION */

static boolean __STATE_PROG_V0;
static boolean __STATE_PROG_V1;

/* INPUT FUNCTIONS */

void STATE_PROG_I_e () {
__STATE_PROG_V0 = _true;
}
void STATE_PROG_I_f () {
__STATE_PROG_V1 = _true;
}

/* PRESENT SIGNAL TESTS */

#define __STATE_PROG_A1 \
__STATE_PROG_V0
#define __STATE_PROG_A2 \
__STATE_PROG_V1

/* ASSIGNMENTS */

#define __STATE_PROG_A3 \
__STATE_PROG_V0 = _false
#define __STATE_PROG_A4 \
__STATE_PROG_V1 = _false

/* PROCEDURE CALLS */

#define __STATE_PROG_A5 \
inA()
#define __STATE_PROG_A6 \
outA()
#define __STATE_PROG_A7 \
outA()
#define __STATE_PROG_A8 \
outB()
#define __STATE_PROG_A9 \
inC1()
#define __STATE_PROG_A10 \
inC2()
#define __STATE_PROG_A11 \
prE()
```

```
#define __STATE_PROG_A12 \
outD()
#define __STATE_PROG_A13 \
inE_PROG_PROG_2_D_a_PROG_PROG_2_D_E_b()
#define __STATE_PROG_A14 \
outE()
#define __STATE_PROG_A15 \
outF_PROG_PROG_2_D_F_a_PROG_PROG_2_D_F_c()

/* FUNCTIONS RETURNING NUMBER OF EXEC */

int STATE_PROG_number_of_execs () {
return (0);
}

/* AUTOMATON (STATE ACTION-TREES) */

static void __STATE_PROG__reset_input () {
__STATE_PROG_V0 = _false;
__STATE_PROG_V1 = _false;
}

/* REDEFINABLE BIT TYPE */

#ifndef __SSC_BIT_TYPE_DEFINED
typedef char __SSC_BIT_TYPE;
#endif

/* REGISTER VARIABLES */

static __SSC_BIT_TYPE __STATE_PROG_R[22] = {_true,
 _false, _false, _false, _false, _false, _false, _false,
 _false, _false, _false, _false, _false, _false, _false,
 _false, _false, _false, _false, _false, _false, _false };

/* AUTOMATON ENGINE */

int STATE_PROG () {
/* AUXILIARY VARIABLES */

static __SSC_BIT_TYPE E[106];
E[0] = (
__STATE_PROG_A2);
E[1] = __STATE_PROG_R[0];
E[2] = __STATE_PROG_R[4]&&!(__STATE_PROG_R[0]);
E[3] = E[2]&&(
__STATE_PROG_A2);
E[4] = E[3]||__STATE_PROG_R[0];
E[5] = E[4];
E[6] = __STATE_PROG_R[21]&&!(__STATE_PROG_R[0]);
E[7] = E[6]&&(
__STATE_PROG_A1);
E[8] = E[7];
E[9] = E[7]||__STATE_PROG_R[0];
E[10] = E[9];
E[11] = __STATE_PROG_R[11]&&!(__STATE_PROG_R[0]);
E[12] = (
__STATE_PROG_A1)&&E[11];
if (E[12]) {
__STATE_PROG_A11;
}
if (E[12]) {
__STATE_PROG_A12;
}
E[13] = E[12];
E[14] = __STATE_PROG_R[17]||__STATE_PROG_R[16];
E[15] = __STATE_PROG_R[18]||E[14];
E[16] = __STATE_PROG_R[14]||__STATE_PROG_R[13];
E[17] = __STATE_PROG_R[15]||E[16];
E[18] = __STATE_PROG_R[12]||E[15]||E[17];
E[19] = E[18]&&!(__STATE_PROG_R[0]);
E[20] = E[19]&&!(E[12]);
E[21] = E[20]&&__STATE_PROG_R[18];
E[22] = E[21]&&(
__STATE_PROG_A2);
if (E[22]) {
__STATE_PROG_A15;
}
E[23] = E[22];
E[24] = E[20]&&__STATE_PROG_R[15];
E[25] = (
__STATE_PROG_A2)&&E[24];
if (E[25]) {
__STATE_PROG_A14;
}
E[26] = E[25];
E[27] = E[12];
E[28] = __STATE_PROG_R[7]&&!(__STATE_PROG_R[0]);
```

```
E[29] = E[28]&&(
__STATE_PROG_A2);
if (E[29]) {
__STATE_PROG_A8;
}
E[28] = E[28]&&!((
__STATE_PROG_A2));
E[30] = E[28]&&(
__STATE_PROG_A1);
E[31] = E[29]||E[30];
E[32] = E[31];
E[33] = E[31];
if (E[3]) {
__STATE_PROG_A6;
}
E[2] = E[2]&&!((
__STATE_PROG_A2));
E[34] = E[2]&&(
__STATE_PROG_A1);
if (E[34]) {
__STATE_PROG_A7;
}
E[35] = E[3]||E[34];
E[36] = E[35];
E[37] = E[34];
E[38] = (
__STATE_PROG_A1);
E[39] = _false;
E[40] = _false;
E[41] = E[7];
E[42] = _false;
E[43] = E[22];
E[44] = _false;
E[45] = E[25];
E[46] = E[20]&&__STATE_PROG_R[14];
E[19] = E[19]&&E[12];
E[47] = (E[19]&&__STATE_PROG_R[14])||(E[46]&&E[25]&&__STATE_PROG_R[14]);
E[48] = E[20]&&__STATE_PROG_R[13];
E[48] = (E[46]&&!(E[25]))||(E[48]&&!(E[25])&&__STATE_PROG_R[13])||
   ((E[19]||(E[48]&&E[25]))&&__STATE_PROG_R[13]);
E[16] = (E[17]&&!(E[16]))||E[47]||E[48];
E[24] = !((
__STATE_PROG_A2))&&E[24];
E[24] = (E[19]&&__STATE_PROG_R[15])||E[24];
E[17] = (E[17]&&!(__STATE_PROG_R[15]))||E[24];
E[46] = (E[17]||E[25])&&E[16]&&E[25];
E[49] = E[20]&&__STATE_PROG_R[17];
E[50] = (E[19]&&__STATE_PROG_R[17])||(E[49]&&E[22]&&__STATE_PROG_R[17]);
E[51] = E[20]&&__STATE_PROG_R[16];
E[51] = (E[49]&&!(E[22]))||(E[51]&&!(E[22])&&__STATE_PROG_R[16])||
        (((E[51]&&E[22])||E[19])&&__STATE_PROG_R[16]);
E[14] = (E[15]&&!(E[14]))||E[50]||E[51];
E[21] = E[21]&&!((
__STATE_PROG_A2));
E[21] = (E[19]&&__STATE_PROG_R[18])||E[21];
E[15] = (E[15]&&!(__STATE_PROG_R[18]))||E[21];
E[49] = (E[15]||E[22])&&E[14]&&E[22];
E[52] = E[49]||E[46];
E[53] = E[52];
E[54] = E[25];
E[55] = _false;
E[56] = E[12];
E[57] = __STATE_PROG_R[10]&&!(__STATE_PROG_R[0]);
E[58] = E[57]&&!(E[12]);
E[59] = E[58]||E[22];
E[52] = E[58]||E[52];
E[58] = E[59]&&E[52];
E[60] = E[58]&&(
__STATE_PROG_A2);
if (E[60]) {
__STATE_PROG_A13;
}
E[6] = E[6]&&!((
__STATE_PROG_A1));
E[61] = __STATE_PROG_R[19]&&!(__STATE_PROG_R[0]);
E[62] = __STATE_PROG_R[20]&&!(__STATE_PROG_R[0]);
E[61] = (E[62]&&!(E[7]))||(E[61]&&!(E[7])&&__STATE_PROG_R[19])||
        (E[61]&&E[7]&&__STATE_PROG_R[19]);
E[62] = E[62]&&E[7]&&__STATE_PROG_R[20];
E[63] = __STATE_PROG_R[20]||__STATE_PROG_R[19];
E[64] = E[63]||__STATE_PROG_R[21];
E[63] = (E[64]&&!(E[63]))||E[62]||E[61];
E[65] = (E[64]&&!(__STATE_PROG_R[21]))||E[6];
E[7] = (E[65]||E[7])&&E[63]&&E[7];
E[66] = E[58]&&!((
__STATE_PROG_A2));
E[66] = E[66]||E[60];
E[52] = !(E[59])&&E[52];
```

```
E[67] = E[52]&&E[25];
E[52] = E[52]&&!(E[25]);
E[19] = (E[20]&&__STATE_PROG_R[12])||(E[19]&&__STATE_PROG_R[12]);
E[14] = ((E[24]||E[47]||E[48])&&E[17]&&E[16])||((E[21]||E[50]||E[51])&&
        E[15]&&E[14])||E[19]||E[52]||E[67]||E[66];
E[57] = E[57]&&E[12]&&__STATE_PROG_R[10];
E[18] = __STATE_PROG_R[10]||E[18];
E[15] = E[18]||__STATE_PROG_R[11];
E[18] = (E[15]&&!(E[18]))||E[57]||E[14];
E[11] = !((
__STATE_PROG_A1))&&E[11];
E[16] = E[11]||(E[15]&&!(__STATE_PROG_R[11]));
E[17] = (E[16]||E[12])&&E[18]&&E[12];
E[20] = E[7]||E[17];
E[68] = E[20]||__STATE_PROG_R[0];
E[69] = !(E[9])&&E[68];
E[70] = E[69]&&E[12];
E[71] = E[70];
E[68] = E[9]&&E[68];
E[49] = E[17]||E[49]||E[49]||E[49];
E[9] = !(E[17])&&E[67];
E[72] = E[67];
E[67] = E[67];
E[73] = E[68]&&(
__STATE_PROG_A2);
if (E[73]) {
__STATE_PROG_A13;
}
E[46] = E[17]||E[46]||E[46]||E[46];
E[74] = !(E[17])&&E[66];
E[75] = E[68]&&!((
__STATE_PROG_A2));
E[75] = E[75]||E[73];
E[66] = E[75]||E[66];
E[58] = E[68]||E[58];
E[59] = E[68]||E[59];
E[76] = E[68];
E[20] = E[20];
E[69] = E[69]&&!(E[12]);
E[77] = __STATE_PROG_R[9]&&!(__STATE_PROG_R[0]);
E[78] = E[70];
E[79] = E[68];
E[80] = _false;
E[81] = E[31];
E[82] = _false;
E[83] = E[35];
E[28] = E[28]&&!((
__STATE_PROG_A1));
E[84] = __STATE_PROG_R[5]&&!(__STATE_PROG_R[0]);
E[85] = __STATE_PROG_R[6]&&!(__STATE_PROG_R[0]);
E[84] = (!(E[31])&&E[85])||(!(E[31])&&E[84]&&__STATE_PROG_R[5])||
        (E[31]&&E[84]&&__STATE_PROG_R[5]);
E[85] = E[31]&&E[85]&&__STATE_PROG_R[6];
E[86] = __STATE_PROG_R[6]||__STATE_PROG_R[5];
E[87] = E[86]||__STATE_PROG_R[7];
E[86] = (E[87]&&!(E[86]))||E[85]||E[84];
E[88] = (E[87]&&!(__STATE_PROG_R[7]))||E[28];
E[30] = E[31]&&(E[88]||E[29]||E[30])&&E[86];
E[89] = __STATE_PROG_R[2]&&!(__STATE_PROG_R[0]);
E[90] = __STATE_PROG_R[3]&&!(__STATE_PROG_R[0]);
E[89] = (!(E[35])&&E[90])||(!(E[35])&&E[89]&&__STATE_PROG_R[2])||
        (E[35]&&E[89]&&__STATE_PROG_R[2]);
E[90] = E[35]&&E[90]&&__STATE_PROG_R[3];
E[91] = __STATE_PROG_R[3]||__STATE_PROG_R[2];
E[92] = E[91]||__STATE_PROG_R[4];
E[91] = (E[92]&&!(E[91]))||E[90]||E[89];
E[2] = E[2]&&!((
__STATE_PROG_A1));
E[93] = (E[92]&&!(__STATE_PROG_R[4]))||E[2];
E[35] = E[35]&&(E[93]||E[3]||E[34])&&E[91];
E[94] = E[30]||E[35];
E[95] = E[94]||__STATE_PROG_R[0];
E[96] = !(E[4])&&E[95];
E[97] = E[96]&&E[34];
E[98] = E[97];
E[95] = E[4]&&E[95];
E[4] = E[95]&&__STATE_PROG_R[0];
if (E[4]) {
__STATE_PROG_A5;
}
E[99] = (E[95]&&!(__STATE_PROG_R[0]))||E[4];
E[100] = E[99];
E[94] = E[94];
E[96] = E[96]&&!(E[34]);
E[101] = !(E[31])&&E[96];
E[102] = __STATE_PROG_R[1]&&!(__STATE_PROG_R[0]);
E[96] = E[31]&&E[96];
E[31] = E[96]&&!((
```

```
__STATE_PROG_A2));
E[103] = E[31]&&(
__STATE_PROG_A1);
if (E[103]) {
__STATE_PROG_A10;
}
E[104] = E[96]&&(
__STATE_PROG_A2);
if (E[104]) {
__STATE_PROG_A9;
}
if (_false) {
__STATE_PROG_A10;
}
if (_false) {
__STATE_PROG_A9;
}
E[31] = E[31]&&!((
__STATE_PROG_A1));
E[31] = E[31]||E[104]||E[103];
E[105] = __STATE_PROG_R[8]&&!(__STATE_PROG_R[0]);
E[86] = ((E[2]||E[90]||E[89])&&E[93]&&E[91])||((E[28]||E[85]||E[84])&&
        E[88]&&E[86])||E[102]||E[105]||E[101]||E[97]||E[99]||E[31];
E[63] = ((E[11]||E[57]||E[14])&&E[16]&&E[18])||((E[6]||E[62]||E[61])&&
        E[65]&&E[63])||E[77]||E[69]||E[70]||((E[75]||E[68])&&E[75]&&E[68]);
E[92] = __STATE_PROG_R[1]||__STATE_PROG_R[8]||E[87]||E[92];
E[15] = __STATE_PROG_R[9]||E[64]||E[15];
E[64] = E[92]||E[15];
E[63] = ((E[64]&&!(E[92]))||E[86])&&((E[64]&&!(E[15]))||E[63])&&(E[86]||E[63]);
E[96] = E[96];
E[86] = E[97];
E[95] = E[95];
E[15] = _false;
__STATE_PROG_R[0] = _false;
__STATE_PROG_R[1] = E[102]||E[101];
__STATE_PROG_R[2] = E[99]||(!(E[35])&&E[89]);
__STATE_PROG_R[3] = !(E[35])&&E[90];
__STATE_PROG_R[4] = E[99]||(!(E[35])&&E[2]);
__STATE_PROG_R[5] = E[97]||(!(E[30])&&E[84]);
__STATE_PROG_R[6] = !(E[30])&&E[85];
__STATE_PROG_R[7] = E[97]||(!(E[30])&&E[28]);
__STATE_PROG_R[8] = E[105]||E[31];
__STATE_PROG_R[9] = E[77]||E[69];
__STATE_PROG_R[10] = !(E[17])&&E[57];
__STATE_PROG_R[11] = E[68]||(!(E[17])&&E[11]);
__STATE_PROG_R[12] = (!(E[17])&&E[19])||(!(E[17])&&E[52]);
__STATE_PROG_R[13] = E[75]||E[74]||(!(E[46])&&E[48]);
__STATE_PROG_R[14] = !(E[46])&&E[47];
__STATE_PROG_R[15] = E[75]||E[74]||(!(E[46])&&E[24]);
__STATE_PROG_R[16] = E[9]||(!(E[49])&&E[51]);
__STATE_PROG_R[17] = !(E[49])&&E[50];
__STATE_PROG_R[18] = E[9]||(!(E[49])&&E[21]);
__STATE_PROG_R[19] = E[70]||(!(E[7])&&E[61]);
__STATE_PROG_R[20] = !(E[7])&&E[62];
__STATE_PROG_R[21] = E[70]||(!(E[7])&&E[6]);
__STATE_PROG__reset_input();
return E[63];
}

/* AUTOMATON RESET */

int STATE_PROG_reset () {
__STATE_PROG_R[0] = _true;
__STATE_PROG_R[1] = _false;
__STATE_PROG_R[2] = _false;
 [...]
__STATE_PROG_R[21] = _false;
__STATE_PROG__reset_input();
return 0;
}
```

## A.4.2 Esterel-Compiler Output Optimized for Memory Efficiency

```
/* REGISTER VARIABLES */

struct REGISTERS {
   unsigned int R0:1;
   unsigned int R1:1;
   unsigned int R2:1;
 [...]
   unsigned int R21:1;
} __STATE_PROG_bitR;

```

```
/* AUTOMATON ENGINE */

int STATE_PROG() {
/* AUXILIARY VARIABLES */

static struct {
  unsigned int E0:1;
  unsigned int E1:1;
  unsigned int E2:1;
 [...]
  unsigned int E104:1;
  unsigned int E105:1;
} bitE;

bitE.E0 = (__STATE_PROG_A2);
bitE.E1 = __STATE_PROG_bitR.R0;
bitE.E2 = __STATE_PROG_bitR.R4&&!(__STATE_PROG_bitR.R0);
bitE.E3 = bitE.E2&&(__STATE_PROG_A2);
bitE.E4 = bitE.E3||__STATE_PROG_bitR.R0;
bitE.E5 = bitE.E4;
bitE.E6 = __STATE_PROG_bitR.R21&&!(__STATE_PROG_bitR.R0);
bitE.E7 = bitE.E6&&(__STATE_PROG_A1);
bitE.E8 = bitE.E7;
bitE.E9 = bitE.E7||__STATE_PROG_bitR.R0;
bitE.E10 = bitE.E9;
bitE.E11 = __STATE_PROG_bitR.R11&&!(__STATE_PROG_bitR.R0);
bitE.E12 = (__STATE_PROG_A1)&&bitE.E11;
if (bitE.E12) {
__STATE_PROG_A11;
}
if (bitE.E12) {
__STATE_PROG_A12;
}
bitE.E13 = bitE.E12;
bitE.E14 = __STATE_PROG_bitR.R17||__STATE_PROG_bitR.R16;
bitE.E15 = __STATE_PROG_bitR.R18||bitE.E14;
bitE.E16 = __STATE_PROG_bitR.R14||__STATE_PROG_bitR.R13;
bitE.E17 = __STATE_PROG_bitR.R15||bitE.E16;
bitE.E18 = __STATE_PROG_bitR.R12||bitE.E15||bitE.E17;
bitE.E19 = bitE.E18&&!(__STATE_PROG_bitR.R0);
bitE.E20 = bitE.E19&&!(bitE.E12);
bitE.E21 = bitE.E20&&__STATE_PROG_bitR.R18;
bitE.E22 = bitE.E21&&(__STATE_PROG_A2);
if (bitE.E22) {
__STATE_PROG_A15;
}
bitE.E23 = bitE.E22;
bitE.E24 = bitE.E20&&__STATE_PROG_bitR.R15;
bitE.E25 = (__STATE_PROG_A2)&&bitE.E24;
if (bitE.E25) {
__STATE_PROG_A14;
}
bitE.E26 = bitE.E25;
bitE.E27 = bitE.E12;
bitE.E28 = __STATE_PROG_bitR.R7&&!(__STATE_PROG_bitR.R0);
bitE.E29 = bitE.E28&&(__STATE_PROG_A2);
if (bitE.E29) {
__STATE_PROG_A8;
}
bitE.E28 = bitE.E28&&!((__STATE_PROG_A2));
bitE.E30 = bitE.E28&&(__STATE_PROG_A1);
bitE.E31 = bitE.E29||bitE.E30;
bitE.E32 = bitE.E31;
bitE.E33 = bitE.E31;
if (bitE.E3) {
__STATE_PROG_A6;
}
bitE.E2 = bitE.E2&&!((__STATE_PROG_A2));
bitE.E34 = bitE.E2&&(__STATE_PROG_A1);
if (bitE.E34) {
__STATE_PROG_A7;
}
bitE.E35 = bitE.E3||bitE.E34;
bitE.E36 = bitE.E35;
bitE.E37 = bitE.E34;
bitE.E38 = (__STATE_PROG_A1);
bitE.E39 = _false;
bitE.E40 = _false;
bitE.E41 = bitE.E7;
bitE.E42 = _false;
bitE.E43 = bitE.E22;
bitE.E44 = _false;
bitE.E45 = bitE.E25;
bitE.E46 = bitE.E20&&__STATE_PROG_bitR.R14;
bitE.E19 = bitE.E19&&bitE.E12;
bitE.E47 = (bitE.E19&&__STATE_PROG_bitR.R14)||(bitE.E46&&bitE.E25&&
           __STATE_PROG_bitR.R14);
```

```
387 bitE.E48 = bitE.E20&&__STATE_PROG_bitR.R13;
388 bitE.E48 = (bitE.E46&&!(bitE.E25))||(bitE.E48&&!(bitE.E25)&&
389           __STATE_PROG_bitR.R13)||((bitE.E19||(bitE.E48&&bitE.E25))&&
390           __STATE_PROG_bitR.R13);
391 bitE.E16 = (bitE.E17&&!(bitE.E16))||bitE.E47||bitE.E48;
392 bitE.E24 = !((__STATE_PROG_A2))&&bitE.E24;
393 bitE.E24 = (bitE.E19&&__STATE_PROG_bitR.R15)||bitE.E24;
394 bitE.E17 = (bitE.E17&&!(__STATE_PROG_bitR.R15))||bitE.E24;
395 bitE.E46 = (bitE.E17||bitE.E25)&&bitE.E16&&bitE.E25;
396 bitE.E49 = bitE.E20&&__STATE_PROG_bitR.R17;
397 bitE.E50 = (bitE.E19&&__STATE_PROG_bitR.R17)||(bitE.E49&&bitE.E22&&
398           __STATE_PROG_bitR.R17);
399 bitE.E51 = bitE.E20&&__STATE_PROG_bitR.R16;
400 bitE.E51 = (bitE.E49&&!(bitE.E22))||(bitE.E51&&!(bitE.E22)&&
401           __STATE_PROG_bitR.R16)||(((bitE.E51&&bitE.E22)||bitE.E19)&&
402           __STATE_PROG_bitR.R16);
403 bitE.E14 = (bitE.E15&&!(bitE.E14))||bitE.E50||bitE.E51;
404 bitE.E21 = bitE.E21&&!((__STATE_PROG_A2));
405 bitE.E21 = (bitE.E19&&__STATE_PROG_bitR.R18)||bitE.E21;
406 bitE.E15 = (bitE.E15&&!(__STATE_PROG_bitR.R18))||bitE.E21;
407 bitE.E49 = (bitE.E15||bitE.E22)&&bitE.E14&&bitE.E22;
408 bitE.E52 = bitE.E49||bitE.E46;
409 bitE.E53 = bitE.E52;
410 bitE.E54 = bitE.E25;
411 bitE.E55 = _false;
412 bitE.E56 = bitE.E12;
413 bitE.E57 = __STATE_PROG_bitR.R10&&!(__STATE_PROG_bitR.R0);
414 bitE.E58 = bitE.E57&&!(bitE.E12);
415 bitE.E59 = bitE.E58||bitE.E22;
416 bitE.E52 = bitE.E58||bitE.E52;
417 bitE.E58 = bitE.E59&&bitE.E52;
418 bitE.E60 = bitE.E58&&(__STATE_PROG_A2);
419 if (bitE.E60) {
420 __STATE_PROG_A13;
421 }
422 bitE.E6 = bitE.E6&&!((__STATE_PROG_A1));
423 bitE.E61 = __STATE_PROG_bitR.R19&&!(__STATE_PROG_bitR.R0);
424 bitE.E62 = __STATE_PROG_bitR.R20&&!(__STATE_PROG_bitR.R0);
425 bitE.E61 = (bitE.E62&&!(bitE.E7))||(bitE.E61&&!(bitE.E7)&&
426           __STATE_PROG_bitR.R19)||(bitE.E61&&bitE.E7&&__STATE_PROG_bitR.R19);
427 bitE.E62 = bitE.E62&&bitE.E7&&__STATE_PROG_bitR.R20;
428 bitE.E63 = __STATE_PROG_bitR.R20||__STATE_PROG_bitR.R19;
429 bitE.E64 = bitE.E63||__STATE_PROG_bitR.R21;
430 bitE.E63 = (bitE.E64&&!(bitE.E63))||bitE.E62||bitE.E61;
431 bitE.E65 = (bitE.E64&&!(__STATE_PROG_bitR.R21))||bitE.E6;
432 bitE.E7 = (bitE.E65||bitE.E7)&&bitE.E63&&bitE.E7;
433 bitE.E66 = bitE.E58&&!((__STATE_PROG_A2));
434 bitE.E66 = bitE.E66||bitE.E60;
435 bitE.E52 = !(bitE.E59)&&bitE.E52;
436 bitE.E67 = bitE.E52&&bitE.E25;
437 bitE.E52 = bitE.E52&&!(bitE.E25);
438 bitE.E19 = (bitE.E20&&__STATE_PROG_bitR.R12)||(bitE.E19&&__STATE_PROG_bitR.R12);
439 bitE.E14 = ((bitE.E24||bitE.E47||bitE.E48)&&bitE.E17&&bitE.E16)||
440           ((bitE.E21||bitE.E50||bitE.E51)&&bitE.E15&&bitE.E14)||
441           bitE.E19||bitE.E52||bitE.E67||bitE.E66;
442 bitE.E57 = bitE.E57&&bitE.E12&&__STATE_PROG_bitR.R10;
443 bitE.E18 = __STATE_PROG_bitR.R10||bitE.E18;
444 bitE.E15 = bitE.E18||__STATE_PROG_bitR.R11;
445 bitE.E18 = (bitE.E15&&!(bitE.E18))||bitE.E57||bitE.E14;
446 bitE.E11 = !((__STATE_PROG_A1))&&bitE.E11;
447 bitE.E16 = bitE.E11||(bitE.E15&&!(__STATE_PROG_bitR.R11));
448 bitE.E17 = (bitE.E16||bitE.E12)&&bitE.E18&&bitE.E12;
449 bitE.E20 = bitE.E7||bitE.E17;
450 bitE.E68 = bitE.E20||__STATE_PROG_bitR.R0;
451 bitE.E69 = !(bitE.E9)&&bitE.E68;
452 bitE.E70 = bitE.E69&&bitE.E12;
453 bitE.E71 = bitE.E70;
454 bitE.E68 = bitE.E9&&bitE.E68;
455 bitE.E49 = bitE.E17||bitE.E49||bitE.E49||bitE.E49;
456 bitE.E9 = !(bitE.E17)&&bitE.E67;
457 bitE.E72 = bitE.E67;
458 bitE.E67 = bitE.E67;
459 bitE.E73 = bitE.E68&&(__STATE_PROG_A2);
460 if (bitE.E73) {
461 __STATE_PROG_A13;
462 }
463 bitE.E46 = bitE.E17||bitE.E46||bitE.E46||bitE.E46;
464 bitE.E74 = !(bitE.E17)&&bitE.E66;
465 bitE.E75 = bitE.E68&&!((__STATE_PROG_A2));
466 bitE.E75 = bitE.E75||bitE.E73;
467 bitE.E66 = bitE.E75||bitE.E66;
468 bitE.E58 = bitE.E68||bitE.E58;
469 bitE.E59 = bitE.E68||bitE.E59;
470 bitE.E76 = bitE.E68;
471 bitE.E20 = bitE.E20;
472 bitE.E69 = bitE.E69&&!(bitE.E12);
473 bitE.E77 = __STATE_PROG_bitR.R9&&!(__STATE_PROG_bitR.R0);
474 bitE.E78 = bitE.E70;
```

```
475 bitE.E79 = bitE.E68;
476 bitE.E80 = _false;
477 bitE.E81 = bitE.E31;
478 bitE.E82 = _false;
479 bitE.E83 = bitE.E35;
480 bitE.E28 = bitE.E28&&!((__STATE_PROG_A1));
481 bitE.E84 = __STATE_PROG_bitR.R5&&!(__STATE_PROG_bitR.R0);
482 bitE.E85 = __STATE_PROG_bitR.R6&&!(__STATE_PROG_bitR.R0);
483 bitE.E84 = (!(bitE.E31)&&bitE.E85)||(!(bitE.E31)&&bitE.E84&&
484          __STATE_PROG_bitR.R5)||(bitE.E31&&bitE.E84&&__STATE_PROG_bitR.R5);
485 bitE.E85 = bitE.E31&&bitE.E85&&__STATE_PROG_bitR.R6;
486 bitE.E86 = __STATE_PROG_bitR.R6||__STATE_PROG_bitR.R5;
487 bitE.E87 = bitE.E86||__STATE_PROG_bitR.R7;
488 bitE.E86 = (bitE.E87&&!(bitE.E86))||bitE.E85||bitE.E84;
489 bitE.E88 = (bitE.E87&&!(__STATE_PROG_bitR.R7))||bitE.E28;
490 bitE.E30 = bitE.E31&&(bitE.E88||bitE.E29||bitE.E30)&&bitE.E86;
491 bitE.E89 = __STATE_PROG_bitR.R2&&!(__STATE_PROG_bitR.R0);
492 bitE.E90 = __STATE_PROG_bitR.R3&&!(__STATE_PROG_bitR.R0);
493 bitE.E89 = (!(bitE.E35)&&bitE.E90)||(!(bitE.E35)&&bitE.E89&&
494          __STATE_PROG_bitR.R2)||(bitE.E35&&bitE.E89&&__STATE_PROG_bitR.R2);
495 bitE.E90 = bitE.E35&&bitE.E90&&__STATE_PROG_bitR.R3;
496 bitE.E91 = __STATE_PROG_bitR.R3||__STATE_PROG_bitR.R2;
497 bitE.E92 = bitE.E91||__STATE_PROG_bitR.R4;
498 bitE.E91 = (bitE.E92&&!(bitE.E91))||bitE.E90||bitE.E89;
499 bitE.E2 = bitE.E2&&!((__STATE_PROG_A1));
500 bitE.E93 = (bitE.E92&&!(__STATE_PROG_bitR.R4))||bitE.E2;
501 bitE.E35 = bitE.E35&&(bitE.E93||bitE.E3||bitE.E34)&&bitE.E91;
502 bitE.E94 = bitE.E30||bitE.E35;
503 bitE.E95 = bitE.E94||__STATE_PROG_bitR.R0;
504 bitE.E96 = !(bitE.E4)&&bitE.E95;
505 bitE.E97 = bitE.E96&&bitE.E34;
506 bitE.E98 = bitE.E97;
507 bitE.E95 = bitE.E4&&bitE.E95;
508 bitE.E4 = bitE.E95&&__STATE_PROG_bitR.R0;
509 if (bitE.E4) {
510 __STATE_PROG_A5;
511 }
512 bitE.E99 = (bitE.E95&&!(__STATE_PROG_bitR.R0))||bitE.E4;
513 bitE.E100 = bitE.E99;
514 bitE.E94 = bitE.E94;
515 bitE.E96 = bitE.E96&&!(bitE.E34);
516 bitE.E101 = !(bitE.E31)&&bitE.E96;
517 bitE.E102 = __STATE_PROG_bitR.R1&&!(__STATE_PROG_bitR.R0);
518 bitE.E96 = bitE.E31&&bitE.E96;
519 bitE.E31 = bitE.E96&&!((__STATE_PROG_A2));
520 bitE.E103 = bitE.E31&&(__STATE_PROG_A1);
521 if (bitE.E103) {
522 __STATE_PROG_A10;
523 }
524 bitE.E104 = bitE.E96&&(__STATE_PROG_A2);
525 if (bitE.E104) {
526 __STATE_PROG_A9;
527 }
528 if (_false) {
529 __STATE_PROG_A10;
530 }
531 if (_false) {
532 __STATE_PROG_A9;
533 }
534 bitE.E31 = bitE.E31&&!((__STATE_PROG_A1));
535 bitE.E31 = bitE.E31||bitE.E104||bitE.E103;
536 bitE.E105 = __STATE_PROG_bitR.R8&&!(__STATE_PROG_bitR.R0);
537 bitE.E86 = ((bitE.E2||bitE.E90||bitE.E89)&&bitE.E93&&bitE.E91)||
538          ((bitE.E28||bitE.E85||bitE.E84)&&bitE.E88&&bitE.E86)||
539          bitE.E102||bitE.E105||bitE.E101||bitE.E97||bitE.E99||bitE.E31;
540 bitE.E63 = ((bitE.E11||bitE.E57||bitE.E14)&&bitE.E16&&bitE.E18)||
541          ((bitE.E6||bitE.E62||bitE.E61)&&bitE.E65&&bitE.E63)||bitE.E77||
542          bitE.E69||bitE.E70||((bitE.E75||bitE.E68)&&bitE.E75&&bitE.E68);
543 bitE.E92 = __STATE_PROG_bitR.R1||__STATE_PROG_bitR.R8||bitE.E87||bitE.E92;
544 bitE.E15 = __STATE_PROG_bitR.R9||bitE.E64||bitE.E15;
545 bitE.E64 = bitE.E92||bitE.E15;
546 bitE.E63 = ((bitE.E64&&!(bitE.E92))||bitE.E86)&&((bitE.E64&&!(bitE.E15))||
547          bitE.E63)&&(bitE.E86||bitE.E63);
548 bitE.E96 = bitE.E96;
549 bitE.E86 = bitE.E97;
550 bitE.E95 = bitE.E95;
551 bitE.E15 = _false;
552 __STATE_PROG_bitR.R0 = _false;
553 __STATE_PROG_bitR.R1 = bitE.E102||bitE.E101;
554 __STATE_PROG_bitR.R2 = bitE.E99||(!(bitE.E35)&&bitE.E89);
555 __STATE_PROG_bitR.R3 = !(bitE.E35)&&bitE.E90;
556 __STATE_PROG_bitR.R4 = bitE.E99||(!(bitE.E35)&&bitE.E2);
557 __STATE_PROG_bitR.R5 = bitE.E97||(!(bitE.E30)&&bitE.E84);
558 __STATE_PROG_bitR.R6 = !(bitE.E30)&&bitE.E85;
559 __STATE_PROG_bitR.R7 = bitE.E97||(!(bitE.E30)&&bitE.E28);
560 __STATE_PROG_bitR.R8 = bitE.E105||bitE.E31;
561 __STATE_PROG_bitR.R9 = bitE.E77||bitE.E69;
562 __STATE_PROG_bitR.R10 = !(bitE.E17)&&bitE.E57;
```

```
__STATE_PROG_bitR.R11 = bitE.E68||(!(bitE.E17)&&bitE.E11);
__STATE_PROG_bitR.R12 = (!(bitE.E17)&&bitE.E19)||(!(bitE.E17)&&bitE.E52);
__STATE_PROG_bitR.R13 = bitE.E75||bitE.E74||(!(bitE.E46)&&bitE.E48);
__STATE_PROG_bitR.R14 = !(bitE.E46)&&bitE.E47;
__STATE_PROG_bitR.R15 = bitE.E75||bitE.E74||(!(bitE.E46)&&bitE.E24);
__STATE_PROG_bitR.R16 = bitE.E9||(!(bitE.E49)&&bitE.E51);
__STATE_PROG_bitR.R17 = !(bitE.E49)&&bitE.E50;
__STATE_PROG_bitR.R18 = bitE.E9||(!(bitE.E49)&&bitE.E21);
__STATE_PROG_bitR.R19 = bitE.E70||(!(bitE.E7)&&bitE.E61);
__STATE_PROG_bitR.R20 = !(bitE.E7)&&bitE.E62;
__STATE_PROG_bitR.R21 = bitE.E70||(!(bitE.E7)&&bitE.E6);
__STATE_PROG__reset_input();
return bitE.E63;
}

/* AUTOMATON RESET */

int STATE_PROG_reset () {
__STATE_PROG_bitR.R0 = 1;
__STATE_PROG_bitR.R1 = 0;
__STATE_PROG_bitR.R2 = 0;

 [...]

__STATE_PROG_bitR.R21 = 0;
__STATE_PROG__reset_input();
return 0;
}
```

# A.5 Compilation

## A.5.1 Example Compilation for the BTnode Emulation

```
kasten@laptop ~/projects/kasten/src/osm/osmc2/test > make # #
generating esterel-files from test.osm # java OSM test.osm Checking
... done.  Writing Esterel code and C include file to "test.strl" and
"test.h" ... done.  # # generating c-files from test.strl #
C:\PROGRA~1\Esterel/bin/esterel.exe -Lc:-ansi -causal test.strl
C:\Cygwin\home\kasten\projects\kasten\src\osm\osmc2\test\10>occ -sscc
-ansi -B "test" "C:\Temp\ssc2356.ssc" gcc -I.. -Wall -c -o prog.o
prog.c prog.c: In function ‘main’: prog.c:74: warning: implicit
declaration of function ‘exit’ prog.c:79: warning: implicit
declaration of function ‘STATE_PROG_I_e’ prog.c:82: warning: implicit
declaration of function ‘STATE_PROG_I_f’ gcc -I.. -Wall -c -o
noncanonical_termio.o ../noncanonical_termio.c gcc -I.. -Wall -c -o
test.o test.c test.c:46: warning: ‘__STATE_PROG_PActionArray’ defined
but not used gcc prog.o test.o noncanonical_termio.o -o prog
kasten@laptop ~/projects/kasten/src/osm/osmc2/test >
```

## A.5.2 Makefile

```
# ------------------------------------------------------------
# files and directories
# ------------------------------------------------------------
TARGET = prog

OSM_SRCS     = test.osm
C_SRCS       = prog.c noncanonical_termio.c
# generated sources
GENE_SRCS    = $(OSM_SRCS:.osm=.strl)
GENC_SRCS    = $(OSM_SRCS:.osm=.c)

SRCS  = $(GENC_SRCS) $(C_SRCS)
OBJS  = $(SRCS:.c=.o)

VPATH = ..

# ------------------------------------------------------------
# programs
# ------------------------------------------------------------
# the esterel compiler requires c:\temp to exist!
ESTERELC = $(ESTEREL)/bin/esterel.exe
# java must be in PATH
JAVA    = java
# OSM.class must be in CLASSPATH
OSM     = $(JAVA) OSM

RM = rm -Rf
MV = mv
CP = cp
ECHO = echo
```

```
31
32 # ------------------------------------------------------------
33 # rules
34 # ------------------------------------------------------------
35 .PHONY : all clean
36
37 # all: prog
38 prog: prog.o test.o noncanonical_termio.o
39 prog.o: test.c
40
41 %.strl: %.osm
42         #
43         # generating esterel-files from $<
44         #
45         $(OSM) $<
46
47 %.c: %.strl
48         #
49         # generating c-files from $<
50         #
51         $(ESTERELC) -Lc:-ansi -causal $<
52
53 clean:
54         $(RM) *~ *.class *.o
55         $(RM) $(GENC_SRCS)
56         $(RM) $(GENE_SRCS)
57         $(RM) $(TARGET) $(TARGET).exe
58         $(RM) $(OSM_SRCS:.osm=.c)
59         $(RM) $(OSM_SRCS:.osm=.h)
60
61 # ------------------------------------------------------------------------
62 # Flags
63 # ------------------------------------------------------------------------
64 # (un)comment as appropriate
65
66 # BTnode
67 # CFLAGS += -I$(BTNODE_INSTALL_DIR) -I.. -Wall
68 # LDLIBS += -L$(BTNODE_INSTALL_DIR)/linux -lBTNsys
69
70 # Linux and Cygwin
71 CFLAGS += -I.. -Wall
72 LDLIBS +=
73
74 LDFLAGS +=
75
76 # ------------------------------------------------------------------------
77 # secondary targets, misc
78 # ------------------------------------------------------------------------
79 .SECONDARY: $(addprefix $(basename $(OSM_SRCS)),.o .strl .c)
80
81 SUFFIXES = .osm .strl
```

# B Implementations of the EnviroTrack Group Management

This Appendix shows the NesC and OSM implementations of EnviroTrack's Group Management in Sections B.1 and B.2, respectively. In order to make both implementations comparable, only control-flow code is shown in the NesC code. Code representing actions has been replaced by placeholders `op1()`, `op2`, etc., other code has been replaced by `[...]`. The implementations of actions are not shown; they are equal in both implementations.

## B.1 NesC Implementation

```
/*
 * Copyright (c) 2004-2006, University of Virginia
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF VIRGINIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE VANDERBILT
 * UNIVERSITY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF VIRGINIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF VIRGINIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
 */

// Authors: Brian Blum, Tian He, Liqian Luo

/* ========================================================
 *
 * ======================================================== */
module GroupManagementM {
   provides {
      interface GroupManagement;
      interface StdControl;
   }
   uses {
      interface RoutingSendByBroadcast as SendMsgByBct;
      interface RoutingReceive as ReceiveRoutingMsg;

      interface TimedLeds;
      interface Random;
      interface Leds;
      interface Localization as Local;
      interface Link;
   }
} /* end module GroupManagementM */

/* ========================================================
 *
 * ======================================================== */
implementation {

   /* variables */
   GMStatus _GMStatus;
   uint16_t _generalTimer;
   // [...]

   /* function prototypes */
   inline result_t GMSend( /* [...]*/, GMMsgType type );
```

```
56  inline void initGMStatus();
57
58  // the following functions replace long defines from the original code
59  inline int wait_random();
60  inline int wait_receive();
61  inline int wait_recruit();
62
63  /* tasks */
64  task void ProcessRecuritMessage();
65
66  /* ------------------------------------------------------- *
67   * Commands
68   * ------------------------------------------------------- */
69  command result_t StdControl.init() {
70     initGMStatus();
71     _generalTimer = 0;
72     // [...]
73     return SUCCESS;
74  }
75
76  command result_t StdControl.start() { /* [...] */ }
77
78  command result_t StdControl.stop() { /* [...] */ }
79
80  command result_t GroupManagement.fireHeartBeat() {
81
82     if( _GMStatus.status == FREE )
83        return SUCCESS;
84
85     if( _generalTimer > 0 )
86        _generalTimer --;
87
88     if( _generalTimer <= 0 ) {
89
90        switch( _GMStatus.status ) {
91
92        case FOLLOWER: {
93           initGMStatus();
94           break;
95        } // end case FOLLOWER
96
97        case MEMBER: {
98           op1();
99           _GMStatus.status = LEADERCANDIDATE;
100          _generalTimer = wait_random();
101          break;
102       } // end case MEMBER
103
104       case NEWCANDIDATE: {
105          GMSend( /* [...] */, CANDIDATE );
106          _generalTimer = wait_recruit();
107
108          if( /* [...] */ ) {
109             op2();
110             _GMStatus.status = LEADERCANDIDATE;
111             _generalTimer = wait_random();
112          }
113          break;
114       } // end case NEWCANDIDATE
115
116       case LEADERCANDIDATE: {
117          op3();
118          GMSend( /* [...] */, CANDIDATE );
119          _GMStatus.status = LEADER;
120          _generalTimer = wait_random();
121          break;
122       } // end case LEADERCANDIDATE
123
124       case LEADER: {
125          GMSend( /* [...] */, RECRUIT );
126          _generalTimer = wait_recruit();
127          op4();
128          break;
129       } // end case LEADER
130
131       case RESIGNINGLEADER: {
132          GMSend( /* [...] */, RESIGN );
133          if( /* [...] */ ) {
134            op5();
135            _generalTimer = wait_recruit();  // self transition
136          } else {
137             op6();
138             initGMStatus(); // transition to FREE
139          }
140          break;
141       } // end case RESIGNLEADER
142
143       default: break;
```

```
144         } // end switch( _GMStatus.status )
145
146      } // end if( _generalTimer <= 0 )
147      return SUCCESS;
148   } // end GroupManagement.fireHeartBeat()
149
150
151   command result_t GroupManagement.join() {
152      switch( _GMStatus.status ) {
153      case FREE: {
154         _GMStatus.status = NEWCANDIDATE;
155         _generalTimer = wait_random();
156         break;
157      }
158
159      case FOLLOWER: {
160         op7();
161         _GMStatus.status = MEMBER;
162         _generalTimer = wait_receive();
163         break;
164      }
165
166      case RESIGNINGLEADER: {
167         GMSend( /* [...] */, RECRUIT );
168         op8();
169         _generalTimer = wait_recruit();
170         _GMStatus.status = LEADER;
171         break;
172      }
173
174      default: break;
175      } // end switch( _GMStatus.status )
176      return SUCCESS;
177   } // end GroupManagement.join()
178
179
180   command result_t GroupManagement.leave() {
181      switch( _GMStatus.status ) {
182
183      case MEMBER: {
184         op9();
185         _GMStatus.status = FOLLOWER;
186         _generalTimer = wait_threshold();
187         break;
188      } // end case MEMBER
189
190      case NEWCANDIDATE: {
191         _GMStatus.status = FREE;
192         break;
193      } // end case NEWCANDIDATE
194
195      case LEADERCANDIDATE: {
196         if( /* [...] */ ) {
197            _GMStatus.status = FREE;
198         } else {
199            _GMStatus.status = FOLLOWER;
200            _generalTimer = wait_threshold();
201         }
202         break;
203      } // end case LEADERCANDIDATE
204
205      case LEADER: {
206         GMSend( /* [...] */, RESIGN );
207         op10();
208         _GMStatus.status = RESIGNINGLEADER;
209         _generalTimer = wait_recruit();
210         break;
211      } // end case LEADER
212
213      default: break; // FREE || FOLLOWER || RESIGNINGLEADER
214      } // end switch( _GMStatus.status )
215
216      return SUCCESS;
217   } // GroupManagement.leave()
218
219   // [...]
220
221   /* ------------------------------------------------------- *
222    * Functions
223    * ------------------------------------------------------- */
224   inline int wait_random()  { /* [...] */ }
225   inline int wait_receive() { /* [...] */ }
226   inline int wait_recruit() { /* [...] */ }
227   // [...]
228
229   /* ------------------------------------------------------- */
230   inline void initGMStatus() {
231      _GMStatus.status = FREE;
```

```
232      op11();
233   }
234
235    /* ------------------------------------------------------- *
236     * TASKS
237     * ------------------------------------------------------- */
238    task void ProcessRecuritMessage() {
239       GMPacket* RxBuffer;
240       op12();
241
242       switch( _GMStatus.status ) {
243
244       case FREE: {
245          if( (RxBuffer->type == RECRUIT) || (RxBuffer->type == RESIGN) ) {
246             op13();
247             _GMStatus.status = FOLLOWER;
248             _generalTimer = wait_threshold();
249          }
250          break;
251       } // end case FREE
252
253       case FOLLOWER: {
254          if( (RxBuffer->type == RECRUIT) || (RxBuffer->type == RESIGN) ) {
255             op14();
256             _generalTimer = wait_threshold();
257          }
258          break;
259       } // end case FOLLOWER
260
261       case MEMBER: {
262          if( RxBuffer->type == RECRUIT ) {
263             op15();
264             _generalTimer = wait_receive();
265          } else if( RxBuffer->type == RESIGN ) {
266             if( /* [...] */ ) {
267                op16();
268                _GMStatus.status = LEADERCANDIDATE;
269                _generalTimer = wait_random();
270             }
271          }
272          break;
273       } // end case MEMBER
274
275       case NEWCANDIDATE: {
276          if( RxBuffer->type == RECRUIT ) {
277             op17();
278             _GMStatus.status = MEMBER;
279             _generalTimer = wait_receive();
280          } else if( RxBuffer->type == CANDIDATE ) {
281             if( /* [...] */ ) {
282                op18();
283                _GMStatus.status = MEMBER;
284                _generalTimer = wait_receive();
285             }
286          }
287          break;
288       } // end case NEWCANDIDATE
289
290       case LEADERCANDIDATE: {
291          if( RxBuffer->type == RECRUIT ) {
292             if( /* [...] */ ) {
293                op19();
294                _GMStatus.status = MEMBER;
295                _generalTimer = wait_receive();
296             }
297          } else if( RxBuffer->type == CANDIDATE ) {
298             if( /* [...] */ ) {
299                op20();
300                _GMStatus.status = MEMBER;
301                _generalTimer = wait_receive();
302             }
303          }
304          break;
305       } // end case LEADERCANDIDATE
306
307       case LEADER: {
308          if( RxBuffer->type == RECRUIT ) {
309             if( /* [...] */ ) {
310                if( /* [...] */ ) {
311                   op21();
312                   _GMStatus.status = MEMBER;
313                   _generalTimer = wait_receive();
314                } else {
315                   GMSend( /* [...] */, RECRUIT );
316                   op22();
317                   _generalTimer = wait_recruit();
318                }
319             }
```

```
320     } else if( (RxBuffer->type == RESIGN) ||
321                (RxBuffer->type == CANDIDATE) ) {
322         if( /* [...] */ ) {
323             GMSend( /* [...] */, RECRUIT );
324             op23();
325             _generalTimer = wait_recruit();
326         }
327       }
328       break;
329     } // end case LEADER
330
331     case RESIGNINGLEADER: {
332       if( (RxBuffer->type == RECRUIT) ||
333           (RxBuffer->type == CANDIDATE) ) {
334           if( /* [...] */ ) {
335               op24();
336               _GMStatus.status = FOLLOWER;
337               _generalTimer = wait_threshold();
338           }
339       }
340       break;
341     } // end case RESIGNINGLEADER
342
343     default: break;
344     } // end switch( _GMStatus.status )
345
346     // [...]
347
348   } // end task ProcessRecuritMessage()
349
350 } // end module implementation
```

## B.2 OSM Implementation

```
template state GROUP_MANAGEMENT {
  onEntry:                      / initialize_timer(), init_group_mgmt();

  initial state FREE {
    onEntry:                    / cancel_timer();
    join                        / /**/           -> NEWCANDIDATE;
    recruit_msg OR resign_msg   / op12(), op13() -> FOLLOWER;
  }

  state FOLLOWER {
    onEntry:                    / reset_timer( wait_threshold() );
    join                        / op7()          -> MEMBER;
    timeout                     / op11()         -> FREE;
    recruit_msg OR resign_msg   / op12(), op14() -> self;
  }

  state MEMBER {
    onEntry:                    / reset_timer( wait_receive() );
    leave                       / op9()          -> FOLLOWER;
    resign_msg[...]             / op16()         -> LEADERCANDIDATE;
    timeout                     / op1()          -> LEADERCANDIDATE;
    recruit_msg                 / op12(), op15() -> self;
  }

  state NEWCANDIDATE {
    onEntry: FREE   -> join     / reset_timer( wait_random() );
    timeout[...]                / op2(), send_candidate_msg() -> LEADERCANDIDATE;
    [else]                      / reset_timer( wait_recruit() );
    leave                       / /* */                      -> FREE;
    recriut_msg                 / op17()                     -> MEMBER;
    candidate_msg[...]          / op12(), op18()             -> MEMBER;
  }

  state LEADERCANDIDATE {
    onEntry:                    /  reset_timer( wait_random() );
    leave[...]                  / /* */                      -> FOLLOWER,
         [else]                 / /* */                      -> FREE;
    timeout                     / op3(), send_candidate_msg() -> LEADER;
    recriut_msg[...]            / op19()                     -> MEMBER;
    candidate_msg[...]          / op12(), op20()             -> MEMBER;
  }

  state LEADER {
    onEntry: LEADERCANDIDATE -> / reset_timer( wait_random() );
    onEntry: RESIGNINGLEADER -> / reset_timer( wait_recruit() );
    onEntry: self            -> / reset_timer( wait_recruit() );
    timeout                     / op4(),  send_recruit_msg() -> self;
    leave                       / op10(), send_resign_msg()  -> RESIGNINGLEADER;
    recruit_msg[... AND ...]    / op12(), op21()             -> MEMBER;
    recruit_msg[... AND !...]   / op12(), op22(), send_recruit_msg() -> self;
    resign_msg OR candidate_msg / op12(), op23(), send_recruit_msg() -> self;
  }

  state RESIGNINGLEADER {
    onEntry:                    / reset_timer( wait_recruit() );
    timeout[...]                / op5(), send_resign_msg()  -> FREE,
           [else]               / op6(), send_resign_msg()  -> self;
    join                        / op8(), send_recruit_msg() -> LEADER;
    recruit_msg[...] OR
      candidate_msg[...]        / op12(), op24()            -> FOLLOWER;
  }

  onPreemption: cancel_timer();
} // end state GROUP_MANAGEMENT
```

# Bibliography


[1] *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, San Francisco, CA, USA, 2003. USENIX.

[2] *SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2003. ACM Press.

[3] *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys 2004)*, Boston, MA, USA, 2004. ACM Press.

[4] *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2004. ACM Press.

[5] *IPSN '05: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, Piscataway, NJ, USA, April 25-27 2005. IEEE Press.

[6] T. Abdelzaher, B. Blum, D. Evans, J. George, S. George, L. Gu, T. He, C. Huang, P. Nagaraddi, S. Son, P. Sorokin, J. Stankovic, and A. Wood. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proc. of 24th International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, March 2004.

[7] Tarek Abdelzaher, John Stankovic, Sang Son, Brian Blum, Tian He, Anthony Wood, and Chanyang Lu. A Communication Architecture and Programming Abstractions for Real-Time Embedded Sensor Networks. In *Workshop on Data Distribution for Real-Time Systems (in conjunction with ICDCS 2003)*, Providence, Rhode Island, 2003. Invited Paper.

[8] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications (WSNA'03)*, pages 50–59. ACM Press, 2003.

[9] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 289–302. USENIX Association, 2002.

[10] Charles André. Syncharts: a visual representation of reactive behaviors. Technical report, I3S, Sophia-Antipolis, France, October 1995.

[11] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA '96*, Lille, France, July 1996.

[12] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. Proactive instructions for furniture assembly. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 351–360, London, UK, 2002. Springer-Verlag.

[13] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[14] Can Basaran, Sebnem Baydere, Giancarlo Bongiovanni, Adam Dunkels, M. Onur Ergin, Laura Marie Feeney, Isa Hacioglu, Vlado Handziski, Andreas Köpke, Maria Lijding, Gaia Maselli, Nirvana Meratnia, Chiara Petrioli, Silvia Santini, Lodewijk van Hoesel, Thiemo Voigt, and Andrea Zanella. Research integration: Platform survey—critical evaluation of research platforms for wireless sensor networks, June 2006. Available from: http://www.embedded-wisents.org/studies/survey_wp2.html.

[15] R. Beckwith, D. Teibel, and P. Bowen. Pervasive computing and proactive agriculture. In *Adjunct Proc. PERVASIVE 2004*, Vienna, Austria, April 2004.

[16] Michael Beigl, Hans-Werner Gellersen, and Albrecht Schmidt. Media cups: Experiences with design and use of computer-augmented everyday objects. *Computer Networks, Special Issue on Pervasive Computing, Elsevier*, 2000.

[17] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[18] Gérard Berry. *The Esterel v5 Language Primer, Version v5 91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, July 2000.

[19] Gérard Berry and the Esterel Team. *The Esterel v5.91 System Manual*. INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, June 2000.

[20] Jan Beutel. *Networked Wireless Embedded Systems: Design and Deployment*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Zurich, Switzerland, 2005.

[21] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping Wireless Sensor Network Applications with BTnodes. In Karl et al. [71], pages 323–338.

[22] Jan Beutel, Oliver Kasten, and Matthias Ringwald. Poster Abstract: BTnodes – a distributed platform for sensor nodes. In *SenSys '03* [2], pages 292–293.

[23] Brian M. Blum, Prashant Nagaraddi, Anthony D. Wood, Tarek F. Abdelzaher, Sang Hyuk Son, and Jack Stankovic. An entity maintenance and connection service for sensor networks. In *MobiSys* [1].

[24] Philippe Bonet. The Hogthrob Project. ESF Exploratory Workshop on Wireless Sensor Networks, ETH Zurich, Zurich, Switzerland, April 1-2, 2004. Available from: http://www.hogthrob.dk/index.htm.

[25] A. Boulis and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys* [1].

[26] Frédéric Boussinot and Robert de Simone. The ESTEREL language. *Proc. of the IEEE*, 79(9):1293–1304, September 1991.

[27] Ed Bryan O'Sullivan. The history of threads, in comp.os.research: Frequently answered questions. Online. Visited 2007-08-03. Available from: http://www.faqs.org/faqs/os-research/part1/preamble.html.

[28] Z. Butler, P. Corke, R. Peterson, and D. Rus. Networked cows: Virtual fences for controlling cows. In *WAMES 2004*, Boston, USA, June 2004.

[29] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. Tinygals: A programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 698–704, New York, NY, USA, 2003. ACM Press.

[30] Sung-Eun Choi and E. Christopher Lewis. A study of common pitfalls in simple multi-threaded programs. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 325–329, New York, NY, USA, 2000. ACM Press.

[31] Intel Corporation. Intel mote. Visited 2007-08-03. Available from: http://www.intel.com/research/exploratory/motes.htm.

[32] National Research Council. *Embedded Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Academy Press, Washington, DC, USA, 2001. Available from: http://www.nap.edu/html/embedded_everywhere/.

[33] Robert Cravotta. Reaching down: 32-bit processors aim for 8 bits. EDN Website (www.edn.com), February 2005. Available from: http://www.edn.com/contents/images/502421.pdf.

[34] Inc. Crossbow Technology. Berkeley motes. Visited 2007-08-03. Available from: http://www.xbow.com/Products/wproductsoverview.aspx.

[35] Christian Decker, Albert Krohn, Michael Beigl, and Tobias Zimmer. The particle computer system. In *IPSN '05* [5], pages 443–448.

[36] Cormac Duffy, Utz Roedig, John Herbert, and Cormac Sreenan. An experimental comparison of event driven and multi-threaded sensor node operating systems. In *PERCOMW '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 267–271, Washington, DC, USA, 2007. IEEE Computer Society.

[37] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Tampa, Florida, USA, November 2004.

[38] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005. Available from: http://www.sics.se/~adam/dunkels05using.pdf.

[39] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. An ultra low-power processor for sensor networks. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 32, pages 27–36, Boston, USA, December 2004. ACM Press.

[40] EnviroTrack. Web page. Visited 2007-08-03. Available from: http://www.cs.uiuc.edu/homes/lluo2/EnviroTrack/.

[41] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270, New York, NY, USA, 1999. ACM Press.

[42] Christian Frank. *Role-based Configuration of Wireless Sensor Networks*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Institute for Pervasive Computing, Zürich, Switzerland, June 2007.

[43] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[44] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Des. Test*, 11(4):44–54, October 1994.

[45] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David E. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM Press, 2003.

[46] Hans-Werner Gellersen, Albrecht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications (MONET)*, 7(5):341–351, October 2002.

[47] ScatterWeb GmbH. Scatterweb. Web page. Visited 2007-08-03. Available from: http://www.scatterweb.com/.

[48] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (SNACK). In *SenSys '04* [4], pages 69–80.

[49] The OMG Object Management Group. *OMG Unified Modeling Language Specification*. Mar 2003. Version 1.5.

[50] Nicolas Halbwachs. Synchronous programming of reactive systems. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 1–16, London, UK, 1998. Springer-Verlag.

[51] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. Sensor network software update management: A survey. *International Journal of Network Management*, 15(4):283–294, July 2005.

[52] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. SOS: A Dynamic Operating System for Sensor Nodes. In *MobiSys 2005*, pages 163–176, Seattle, WA, USA, 2005. ACM Press.

[53] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[54] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, and Bruce Krogh. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys* [3], pages 270–283.

[55] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[56] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8020, Washington, DC, USA, 2000. IEEE Computer Society.

[57] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.

[58] Jason L. Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, June 2004.

[59] Jason L. Hill, Robert Szewczyk, A. Woo, S. Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-IX*, pages 93–104, Cambridge MA, USA, November 2000.

[60] Jason Lester Hill. *System architecture for wireless sensor networks*. PhD thesis, University of California, Berkeley, 2003.

[61] Seth Edward-Austin Hollar. Cots dust, large scale models for smart dust. Master's thesis, University of California, Berkeley, 2000.

[62] Lars Erik Holmquist, Friedemann Mattern, Bernt Schiele, Petteri Alahuhta, Michael Beigl, and Hans-W. Gellersen. Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *Proc. Ubicomp 2001*, pages 116–122, Atlanta, USA, September 2001. Springer-Verlag.

[63] ARGO Homepage. Web page. Visited 2007-08-03. Available from: `http://www.argo.ucsd.edu/`.

[64] Wen Hu, Van Nghia Tran, Nirupama Bulusu, Chun Tung Chou, Sanjay Jha, and Andrew Taylor. The design and evaluation of a hybrid sensor network for Cane-toad monitoring. In *IPSN '05* [5], page 71.

[65] J. P. Hubaux, Th. Gross, J. Y. Le Boudec, and M. Vetterli. Towards self-organized mobile ad hoc networks: the Terminodes project. *IEEE Communications Magazine*, 31(1):118–124, 2001.

[66] Clinton Kelly IV, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor-network asynchronous processor. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 24. IEEE Computer Society, 2003.

[67] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, San Jose, California, USA, October 2002. ACM Press.

[68] J. M. Kahn, R. H. Katz, and Kristofer S. J. Pister. Emerging challenges: Mobile networking for smart dust. *Journal of Communications and Networks*, 2(3):188–196, September 2000.

[69] Cornelia Kappler and Georg Riegel. A real-world, simple wireless sensor network for monitoring electrical energy consumption. In Karl et al. [71], pages 339–352.

[70] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, April 2005.

[71] Holger Karl, Andreas Willig, and Adam Wolisz, editors. *Wireless Sensor Networks, First European Workshop, EWSN 2004, Berlin, Germany, January 19-21, 2004, Proceedings*, volume 2920 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Germany, January 2004. Springer-Verlag.

[72] Oliver Kasten. Programming wireless sensor nodes—a state-based model for complex applications. GI/ITG KuVS Fachgespräch Systemsoftware für Pervasive Computing, Universiät Stuttgart, Germany, October, 14-15 2004.

[73] Oliver Kasten and Marc Langheinrich. First Experiences with Bluetooth in the Smart-Its Distributed Sensor Network. In *Workshop on Ubiqitous Computing and Communication, PACT 2001*, October 2001.

[74] Oliver Kasten and Kay Römer. Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines. In *IPSN '05* [5], pages 45–52.

[75] Tae-Hyung Kim and Seongsoo Hong. State machine based operating system architecture for wireless sensor networks. In *Parallel and Distributed Computing: Applications and Technologies : 5th International Conference, PDCAT*, LNCS 3320, pages 803–806, Singapore, December 2004. Springer-Verlag.

[76] I3S Laboratory. Syncchart. Web page. Visited 2007-03-10. Available from: `http://www.i3s.unice.fr/sports/SyncCharts/`.

[77] Edward A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.

[78] Martin Leopold, Mads Bondo Dydensborg, and Philippe Bonnet. Bluetooth and sensor networks: a reality check. In *SenSys '03* [2], pages 103–113.

[79] Philip Levis and David E. Culler. Maté: A tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 36(5):85–95, December 2002.

[80] J. Liu, M. Chu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing, IEEE*, 2(4):50–62, 2003.

[81] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet. In *MobiSys 2004* [3], pages 256–269.

[82] AXONN LLC. AXTracker. Web page. Visited 2005-10-01. Available from: `http://www.axtracker.com/`.

[83] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI 2002*, Boston, USA, December 2002.

[84] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, June 2003. ACM Press.

[85] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[86] M. Maroti, G. Simon, A. Ledeczi, and J. Sztipanovits. Shooter localization in urban terrain. *Computer*, 37(8):60–61, August 2004.

[87] Ian W. Marshall, Christopher Roadknight, Ibiso Wokoma, and Lionel Sacks. Self-organising sensor networks. In UbiNet 2003, London, UK, September 2003.

[88] Kirk Martinez, Royan Ong, and Jane Hart. Glacsweb: a sensor network for hostile environments. In *Proceedings of The First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pages 81–87, Santa Clara, CA, USA, October 2004.

[89] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, New York, NY, USA, 2006. ACM Press.

[90] W.M. Merrill, F. Newberg, K. Sohrabi, W. Kaiser, and G. Pottie. Collaborative networking requirements for unattended ground sensor systems. In *Aerospace Conference*, March 2003.

[91] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[92] Jogesh K. Muppala. Experience with an embedded systems software course. *SIGBED Rev.*, 2(4):29–33, 2005.

[93] Sanjiv Narayan, Frank Vahid, and Daniel D. Gajski. Translating system specifications to VHDL. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 390–394. IEEE Computer Society Press, 1991.

[94] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Winter Technical Conference*, January 1996. Available from: `http://home.pacbell.net/ouster/threads.ppt`.

[95] Axel Poignè, Matthew Morley, Olivier Maffeïs, Leszek Holenderski, and Reinhard Budde. The Synchronous Approach to Designing Reactive Systems. *Formal Methods in System Design*, 12(2):163–187, 1998.

[96] R. Riem-Vis. Cold chain management using an ultra low power wireless sensor network. In *WAMES 2004*, Boston, USA, June 2004.

[97] Hartmut Ritter, Min Tian, Thiemo Voigt, and Jochen H. Schiller. A highly flexible testbed for studies of ad-hoc network behaviour. In *LCN*, pages 746–752. IEEE Computer Society, 2003.

[98] Kay Römer, Christian Frank, Pedro José Marrón, and Christian Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[99] Shad Roundy, Dan Steingart, Luc Frechette, Paul K. Wright, and Jan M. Rabaey. Power sources for wireless sensor networks. In Karl et al. [71], pages 1–17.

[100] Kay Römer. Tracking real-world phenomena with smart dust. In Karl et al. [71], pages 28–43.

[101] Kay Römer. *Time Synchronization and Localization in Sensor Networks*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Institute for Pervasive Computing, Zürich, Switzerland, May 2005.

[102] Andrew Seitz, Derek Wilson, and Jennifer L. Nielsen. Testing pop-up satellite tags as a tool for identifying critical habitat for Pacific halibut (Hippoglossus stenolepis) in the Gulf of Alaska. Exxon Valdez Oil Spill Restoration Project Final Report (Restoration Project 01478), U.S. Geological Survey, Alaska Biological Science Center, Anchorage, Alaska, September 2002.

[103] Jack Shandle. More for less: Stable future for 8-bit microcontrollers. TechOnLine Website, August 2004. Visited 2007-08-03. Available from: http://www.techonline.com/community/ed_resource/feature_article/36930.

[104] Alan C. Shaw. Communicating real-time state machines. *IEEE Trans. Softw. Eng.*, 18(9):805–816, September 1992.

[105] Saurabh Shukla, Nirupama Bulusu, and Sanjay Jha. Cane-toad Monitoring in Kakadu National Park Using Wireless Sensor Networks. In *Network Research Workshop 2004 (18th APAN Meetings)*, Cairns, Australia, July 2004.

[106] Frank Siegemund. Smart-its on the internet – integrating smart objects into the everyday communication infrastructure, September 2002. Available from: http://www.vs.inf.ethz.ch/publ/papers/smartits-demo-note-siegemund.pdf.

[107] Frank Siegemund, Christian Floerkemeier, and Harald Vogt. The value of handhelds in smart environments. In *Personal and Ubiquitous Computing Journal*. Springer-Verlag, October 2004.

[108] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001.

[109] Gyula Simon, Miklós Marót, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04* [4], pages 1–12.

[110] Karsten Strehl, Lothar Thiele, Matthias Gries, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich. Funstate—an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VSLI) Systems*, 9(4):524–544, 2001.

[111] John Suh and Mike Horton. Powering sensor networks - current technology overview. *IEEE Potentials*, 23(3):35–38, August 2004.

[112] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, 2004.

[113] Esterel Technologies. Esterel Compiler (INRIA Ecoles des Mines Academic Compiler). Web page. Visited 2006-06-22. Available from: http://www.esterel-technologies.com/technology/demos/demos.html.

[114] F. Vahid, S. Narayan, and D. D. Gajski. SpecCharts: A VHDL Front-End for Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, June 1995.

[115] Robert von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 19–24, Lihue, Hawaii, USA, May 2003. The USENIX Association.

[116] H. Wang, L. Yip, D. Maniezzo, J. C. Chen, R. E. Hudson, J. Elson, and K. Yao. A Wireless Time-Synchronized COTS Sensor Platform: Applications to Beamforming. In *In Proceedings of IEEE CAS Workshop on Wireless Communications and Networking*, Pasadena, CA, September 2002.

[117] Hanbiao Wang, Jeremy Elson, Lewis Girod, Deborah Estrin, and Kung Yao. Target classification and localization in habitat monitoring. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003)*, Hong Kong, China, April 2003.

[118] Hanbiao Wang, Deborah Estrin, and Lewis Girod. Preprocessing in a tiered sensor network for habitat monitoring. *EURASIP Journal of Applied Signal Processing (EURASIP JASP)*, (4):392–401, March 2003.

[119] B. A. Warneke, M. D. Scott, B. S. Leibowitz, L. Zhou, C. L. Bellew, J. A. Chediak, J. M. Kahn, B. E. Boser, and Kristofer S. J. Pister. An autonomous 16 cubic mm solar-powered node for distributed wireless sensor networks. In *IEEE Sensors*, Orlando, USA, June 2002.

[120] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer*, 45(1):44–51, January 2001.

[121] Andrzej Wasowski and Peter Sestoft. On the formal semantics of visualSTATE statecharts. Technical Report TR-2002-19, IT University of Copenhagen, Denmark, September 2002.

[122] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 70–84. ACM Press, 2001.

[123] ETH Zurich. BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. Web page. Visited 2007-08-03. Available from: http://www.btnode.ethz.ch/.

[124] The Smart-Its Project. Web page. Visited 2007-08-03. Available from: http://www.smart-its.org.

[125] TinyOS. Web page. Visited 2007-08-03. Available from: http://webs.cs.berkeley.edu/tos/.

# Curriculum Vitae

## Oliver Kasten

**Personal Data**

| | |
|---|---|
| Date of Birth | July 10, 1969 |
| Birthplace | Balingen, Germany |
| Citizenship | German |

**Education**

| | |
|---|---|
| 1976–1981 | *Erich-Kästner-Schule*, Pfungstadt, Germany |
| 1981–1987 | *Holbein-Gymnasium*, Augsburg, Germany |
| 1987–1989 | *Schuldorf Bergstrasse*, Seeheim-Jugenheim, Germany |
| June 1989 | Abitur |
| 1991–1999 | Study of Computer Science at *Technische Universität Darmstadt*, Germany |
| 1998 | Visiting student at *International Computer Science Institute (ICSI)*, Berkeley, USA |
| April 1999 | Diplom Informatiker |
| 2000–2007 | Ph.D. Student at the Department of Computer Science, *ETH Zurich*, Switzerland |

**Civil Service**

| | |
|---|---|
| 1989–1990 | *Malteser Hilfsdienst*, Augsburg, Germany |
| 1990–1991 | Alten- und Pflegeheim *Tannenberg*, Seeheim-Jugenheim, Germany |

**Employment**

| | |
|---|---|
| 1999-2005 | Research Assistant at the Department of Computer Science, *ETH Zurich*, Switzerland |
| 2005-2006 | Software Engineer, *Swissphoto AG*, Regensdorf, Switzerland |
| since 2006 | Senior Researcher, *SAP Research (Schweiz)*, Zurich, Switzerland |