

**SENSID: a Sensor  
Network Situation  
Detector**

Mark Kranz

*This report is submitted as partial fulfilment  
of the requirements for the Honours Programme of the  
School of Computer Science and Software Engineering,  
The University of Western Australia,  
2005*

# Abstract

In the field of environmental monitoring, computer applications are often employed to detect occurrences of complex events patterns with spatial and temporal characteristics. The gathering and delivery of the associated data is typically carried out by deeply embedded systems such as wireless sensor networks. Traditional approaches to event pattern matching involve the communication of sensor data to a central database, where powerful computers can perform offline data mining. However, this approach is only sufficient for passive monitoring, as it does not allow the network to react to events. It also wastes valuable energy by transmitting all data regardless of its relevance.

A novel approach to these problems utilises in-network processing to perform on-line detection of event patterns. Development of a middleware application allows applications to subscribe to situations of interest, whilst ignoring irrelevant data. Inspiration for this method has been drawn from active database systems — a platform often used for building applications that need to respond to complex event patterns.

The prototype system SENSID has been designed and built to illustrate this approach, using Berkeley Mica2 motes and the TinyOS operating system. This system has shown that in-network situation detection is a feasible approach. It has been shown to be capable of expressing a variety of situations, from explosion detection to traffic monitoring. It has also demonstrated acceptable performance in typical environmental monitoring applications, despite hardware limitations of speed and memory.

This paper discusses the challenges encountered in building SENSID, and explores a number of optimisations that help this system suit the sensor network environment architecture.

**Keywords:** Event detection, active sensor networks, event programming language

**CR Categories:** D.3.2, D.4.7, D.1.6

# Acknowledgements

I would like to offer a few words of thanks to a number of people whose time and effort have helped get me through this difficult year.

To my Mum, and all my family and friends, thanks for all their timely distractions. Their efforts have kept me well-fed and entertained, and helped me remain sociable against all odds. To my sweet fiancée Fiona, a very special thanks for all her love and support. Were it not for her constant care and tireless patience, I would never have made it this far. I'd also like to thank my supervisor Rachel, for introducing me to the wonderful world of wireless sensor networks, and for being a dedicated teacher, colleague, and friend.

Finally, I'd like to thank my Dad, whose passion for tinkering, and inquisitive nature led to my interest in computer science. His spirit has guided me through this challenging year, and given me strength when times were tough. All my accomplishments have been a result of his infectious enthusiasm and determination.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 WSN Programming</b>	<b>4</b>
2.1 Operating Systems . . . . .	5
2.2 Language Level Constructs . . . . .	6
2.3 Databases . . . . .	6
2.3.1 TinyDB . . . . .	7
2.3.2 SINA . . . . .	8
2.4 Communication Abstractions . . . . .	8
2.5 Active Sensor Frameworks . . . . .	9
2.5.1 SensorWare . . . . .	9
2.5.2 Maté . . . . .	10
2.5.3 DSWare . . . . .	10
2.6 Summary . . . . .	11
<b>3 Event Detection</b>	<b>13</b>
3.1 Active Databases . . . . .	14
3.2 Event-Condition-Action . . . . .	14
3.3 Composite Events . . . . .	14
3.4 Situation Manager . . . . .	15
3.4.1 Situation Specification . . . . .	16
3.4.2 Notation . . . . .	16
3.4.3 Situation Example . . . . .	19

3.4.4	AMIT Architecture and Operation . . . . .	20
3.5	Our Contribution . . . . .	21
3.6	Project Scope . . . . .	22
<b>4</b>	<b>SENSID Design</b>	<b>23</b>
4.1	Situation Subscription . . . . .	24
4.2	Event Dispatch . . . . .	25
4.3	Lifespan Management . . . . .	26
4.4	Operand Filter . . . . .	27
4.5	Operand Storage . . . . .	28
4.6	Situation Detection . . . . .	29
4.7	Situation Notification . . . . .	30
<b>5</b>	<b>Results: Implementation</b>	<b>31</b>
5.1	Memory Management . . . . .	31
5.2	Component Specific Data Structures and Algorithms . . . . .	33
5.2.1	Situation Definition/Subscription . . . . .	33
5.2.2	Dispatcher . . . . .	35
5.2.3	Lifespan Manager . . . . .	38
5.2.4	Storage . . . . .	39
5.2.5	Detection . . . . .	42
5.2.6	Post-Detection . . . . .	47
<b>6</b>	<b>Results: Statistics and Measurements</b>	<b>48</b>
6.1	Tests . . . . .	48
6.1.1	Implementation Environment . . . . .	48
6.1.2	Event Producers . . . . .	48
6.1.3	Simple Application . . . . .	48
6.1.4	Maté . . . . .	49
6.2	Resource and Performance Statistics . . . . .	50
6.2.1	Code Size . . . . .	50

6.2.2	Code Memory Footprint . . . . .	50
6.2.3	Runtime Memory Usage . . . . .	53
6.2.4	Speed Performance . . . . .	54
6.3	Expressiveness . . . . .	59
6.3.1	Comparison vs AMIT . . . . .	59
6.3.2	Comparison vs Composite Events in WSNs . . . . .	60
<b>7</b>	<b>Discussion and Conclusions</b>	<b>66</b>
<b>A</b>	<b>Original Honours Proposal</b>	<b>75</b>
A.1	Background . . . . .	75
A.2	Aim . . . . .	76
A.2.1	Hypothesis . . . . .	76
A.2.2	Justification . . . . .	77
A.2.3	Goals and Success factors . . . . .	77
A.2.4	Anticipated Accomplishments and Deliverables . . . . .	78
A.3	Method . . . . .	78
A.4	Software and Hardware Requirements . . . . .	79
A.4.1	Required Software . . . . .	79
A.4.2	Required Hardware . . . . .	79
<b>B</b>	<b>Mote Hardware Comparisons</b>	<b>82</b>
<b>C</b>	<b>Communication Abstraction</b>	<b>83</b>
C.0.3	Directed Diffusion . . . . .	83
C.0.4	Abstract Regions . . . . .	83
C.0.5	Virtual Markets . . . . .	85
<b>D</b>	<b>Maté Traffic Example</b>	<b>86</b>

# List of Tables

3.1	Definitions of different situation operators . . . . .	17
3.2	Definitions of situation quantifiers and policies . . . . .	18
6.1	Size of SENSID NesC code . . . . .	51
6.2	Memory costs per component . . . . .	52
6.3	Memory allocation costs per unit, per type . . . . .	53
6.4	System performance per component . . . . .	55
6.5	Number of nodes vs actual detection time . . . . .	57
6.6	Detection performance vs size of solution domain . . . . .	57
6.7	The effect of conditions on actual detection time . . . . .	58
A.1	Project Timeline and Work . . . . .	80

# List of Figures

2.1	Sensorware’s script sharing architecture [10]	9
3.1	Situation diagram notation	19
3.2	Example situation: Plant dehydration risk	20
3.3	The AMIT Architecture [40]	21
4.1	SENSID component architecture	24
4.2	Example dispatcher configuration	26
5.1	Memory management data structure	32
5.2	Situation definition and Dispatcher bindings data structure	36
5.3	Event dispatching algorithm	37
5.4	Simple garbage collection example	39
5.5	Event query algorithm	41
5.6	Simple detection algorithm	43
5.7	Multi-purpose backtracking algorithm	45
5.8	Transformed backtracking algorithm	46
6.1	Design diagram of a simple SENSID application	49
6.2	Design diagram of Maté with SENSID extension	51
6.3	Graph of detection performance vs situation size	56
6.4	Situation diagram for explosion detection	62
6.5	Situation diagram for explosion detection, with additional temporal constraints	63
6.6	Situation diagram for traffic detection	65
B.1	The family of motes and their capabilities	82

## CHAPTER 1

# Introduction

Sensor networks are a tool used in a wide variety of fields, such as environmental monitoring [19, 53, 11], studying animal behaviour [50, 47], controlling robots [8], and improving building techniques [26]. The systems are deeply embedded, with extreme limitations on processing power, memory, bandwidth, and energy usage. Current approaches to these applications involve utilising sensors to capture attributes representing aspects of the real-world, and storing data locally or transmitting to a central store.

However, an invariant among these systems is that sensors alone cannot capture the temporal and spatial characteristics of real-world events. A simple example of such an event is the problem of detecting explosions in an external environment [45]. Since no single sensor can detect the occurrence of an explosion, we must instead make inferences based on the readings of multiple sensors, such as temperature, light, and sound. We must also take into account the temporal characteristics of the events, such as the ordering and time in which they occur since it is unlikely that an explosion occurred if the light occurs minutes after hearing the sound.

There are many examples of this problem in environmental monitoring — from tracking water table levels to finding the likelihood of frost occurring. Many environmental variables need to be monitored, such as humidity, soil moisture, wind, temperature and rain. However, to do this we need to be able to specify the complex relationship that exists between the timing and magnitude of sensor readings, or to find times when events (such as rain) have *not* occurred.

Consider also the problem of detecting when a traffic intersection is congested — this may involve using visual or pressure sensors to detect the presence of cars. However, translating the individual events into a real definition of congestion requires an analysis of the *number* of events that occur during periods of both time and space.

Existing solutions to these problems regularly sample and deliver sensor readings, and run off-line data mining on the central store [27]. The primary limitation

with these approaches is that the network does not have the opportunity to react to events. In addition, there is substantial wastage in the energy and bandwidth of the network, as many data samples that are not required are still transmitted. There are other approaches that deal with this problem, using on-line data aggregation [38], or event pattern matching [45, 37]. However, approaches such as data aggregation are often specific to the type of sensor readings taken, and cannot detect complex patterns, nor react to them. Current event pattern matching models are limited by the type of relations that can be expressed — particularly, they cannot detect when events have *not* occurred during some period, or count the *number* of event occurrences during this period. This is partly due to a limited ability to express the interval during which we are interested in events. Thus, problems such as environmental monitoring and traffic congestion are difficult to specify using these methods.

Research into active databases and middleware have developed the concept of *situations*, which are more expressive and generalise to a wider range of domains [5]. However, it remains to be seen whether these semantics will translate to sensor networks, and whether the constrained hardware of sensor network node can handle the intense computation of situation detection.

The purpose of our research is to investigate the feasibility of using situation semantics on sensor network devices. We have designed and built a SEnsor Network SITUation Detector (SENSID), and using Berkeley mica2 motes and TinyOS, we have studied both its expressive power and performance. In particular, we focus our study on how well the temporal characteristics of situations can be detected, as this is a major limiting factor in the expressiveness existing systems. We address the limitations of the mote hardware, what considerations were taken during development, and what innovations and tradeoffs needed to be made.

The SENSID prototype has shown to be capable of handling specifications for various systems, such as the examples listed above. Performance of situation detection is adequate for most tasks, allowing real-time response to simple situations like explosion detection, and acceptable response to complex situations like environmental monitoring. The current prototype appears unsuited for applications involving complex situations over large domains on single motes. A hybrid approach is suggested that may prove suitable for this problem in future.

The rest of our dissertation is as follows: Chapter 2 outlines the current state of the field of sensor networks, including the target applications and their limitations. Chapter 3 looks at the existing approaches for active databases, goes in-depth into the situation specification, and our discusses goals for translating the semantics to sensor networks. Chapters 4 and 5 cover the design and implementation of SENSID, along with the problems we encountered and how they

were resolved. Chapter 6 presents the results of our implementation, including memory and performance statistics, and comparisons of the expressiveness against current systems. Chapter 7 discusses and summarizes our results, and outlines directions for future work in the field.

## CHAPTER 2

# WSN Programming

The current trend of embedded systems is towards the field of Wireless Sensor Networks (WSN) [28]. Wireless sensor networks have the typical properties of embedded systems — they are deployed in large quantities and are used to engage in the physical world through sensors and actuators. They are also subject to severe constraints, suffering from low speed, space and bandwidth, and are often expected to run with high reliability for very long periods of time.

At present, prices for WSN nodes are prohibitively expensive for mainstream use, often limited by the availability of general purpose sensors and software applications. However, it is likely that as prices drop and the technology matures, sensor networks will become a common type of embedded system.

One of the more challenging issues preventing widespread use of WSN's is finding effective ways to define the behaviour of the network. Many early embedded systems had such extreme resource constraints, that the only feasible way to do this was to directly program individual nodes using assembly language[31]. Even though modern technology has allowed us more freedom with use of memory and processing power, there is still a need for developers to find a balance between efficiency and complexity. To this end, there has been a focus on finding alternative programming techniques, suitable for the development of mainstream applications for WSNs.

Current alternative approaches include operating systems [28], language level constructs [14], database models [9, 27, 48], communication abstractions [29, 54, 39], and active sensor frameworks [10, 34, 37]. This chapter outlines some of these approaches and their various trade-offs.

We will sometimes describe an approach in terms of whether it is data-centric or control-centric. Data-centric frameworks are concerned with the gathering and reporting of data from sensors, but do not use the 'intelligent' aspects of devices to react to that information. Conversely, control-centric approaches are *only* focused on how the system can respond to events, but leaves the user 'out of the loop' when it comes to reporting information. Some approaches adopt a balance

between both perspectives, allowing it to deal with both data management and reactive control.

Most of the approaches we look at are intended for the *current* generation of research sensor network nodes — namely the Berkeley mote family. Motes are a type of COTS (common-off-the-shelf) device, typically featuring a single microchip, including RF communications, non-volatile storage, digital-to-analogue converters, battery, and sensors or sensor ports. Full specifications for motes are shown in Appendix B.

## 2.1 Operating Systems

Code for networking devices is either deployed dynamically or statically. Systems that employ a dynamic approach typically employ a fully-fledged operating system kernel — capable of loading new libraries and applications, and executing system calls to perform low-level functionality. These systems are suited to more powerful devices, such as PDA's and mobile-phones.

Statically linked approaches are more reminiscent of the days of assembly-language embedded system programming — the complete application must be compiled to a binary image, and loaded onto the device. Upgrading portions of code is generally not possible. However, the languages used to program such devices tend to provide some of the functionality of an OS kernel, in the form of pre-built modules and language constructs. Since application modules, OS modules, and OS code base compiles into a single image, applications are smaller and more efficient, making these approaches suitable for less powerful devices like motes.

TinyOS is the de facto standard used to program mote and mote-like devices, providing a statically-linked component architecture [28]. It supports multi-threading, task queuing, and event-driven concurrency management. TinyOS is written in a component based C language, called NesC [24].

Before the development of TinyOS, there were few other frameworks able to adapt to both application and hardware diversity. One notable operating system is pOSEK, which also provides similar concurrent and communication facilities under extreme resource constraints. However, because pOSEK was developed by the OSEK/VDX specification, driven by a consortium of automotive companies and suppliers [46], it tends to be control-centric, and its use has thus been limited to devices with actuators, such as car-electronics and household appliances. TinyOS uses a data-centric approach, suitable for sensing and data-gathering, and has subsequently been designed for wireless sensing applications, such as environmental monitoring.

## 2.2 Language Level Constructs

Since TinyOS uses statically linked images, and the NesC programming language was designed specifically for TinyOS, it is difficult to distinguish between TinyOS and NesC framework features. We will classify features of traditional OS kernels, such as task scheduling and event models to be part of TinyOS.

The NesC programming language uses a component-based framework for application design [24]. In this framework, code modules must declare only the interfaces for what facilities are provided and/or used. Modules may then be ‘wired’ to the appropriate producers and consumers.

This framework works well with the TinyOS paradigm, where components can be easily shared between a variety of hardware and applications, and only component wiring need to be altered. Since the underlying code essentially reduces to a single C file — replacing events, messages and component wiring with appropriate sub-routines/inline statements — the final compiled image is still very small and efficient.

In practise, however, code sharing does not always work. There is often a lack of consensus between component writers, since hardware and applications are highly specialised — preventing simple ‘plug-and-play’ reusability from working. Also, TinyOS is a microprogramming (node-level) approach, making it poorly suited for building applications that manage events and data at a macro (network) level [36].

There are notable extensions to the NesC model, such as TinyGALS [14], that reduce the burden of the programmer to synchronise events and avoid race conditions. While this may improve code quality, it does not help developers standardise module design and interactions, and also adds an extra learning curve.

## 2.3 Databases

Many existing monitoring systems use a centralised storage system for delivering samples from sensor devices to end users. This is often implemented by using a pre-defined data collection method, and using data-warehousing to store and retrieve information through database queries. These systems lack flexibility because they are unable to adapt to new spatial and temporal requirements. They also waste energy in producing and transmitting readings that may not be used.

Bonnet et al proposes the *device database system* as a solution to the problem,

whereby the sensor devices are considered to be a part of the database system [9]. Each device is considered to be a mini-server, able to process queries through its own sensing capabilities. Queries are distributed through the network and resolved on both a local and a global level. SQL semantics are used to define selection and joining operations. The device database COUGAR was one of the first examples of effective macro-programming in wireless sensor networks.

### 2.3.1 TinyDB

One of the shortfalls of the pure database approach is that it only describes data acquisition. The model must be extended further to incorporate control features, such as adaptability, application specific resource management, and use of external actuators.

TinyDB is a more advanced system than the early COUGAR system, and has facilities to cover some of the shortfalls [27]. Additional features include:

**Commands and Triggers** : User created commands and event-based triggers allows the database to perform more complex functions than basic queries, including sensor-calibration, external actuation and fault detection.

**Query Aggregation** : In-network query aggregation allows optimisation of resource usage. Aggregates include spatial minima, maxima and average.

**Extensibility** : Using the TinyDB framework, support can be added for extra sensors; commands and triggers; actuators, and aggregates.

**User Access** : The database can concurrently run multiple queries, including timed queries (epoch), current queries (now), and roaming-user queries.

TinyDB is currently implemented as a TinyOS application, and runs on most mote and mote-like architectures. Like COUGAR, it uses SQL-semantics to accept user queries, and is able to synchronise with a configured PostgreSQL database server.

TinyDB still suffers the shortfall of not being truly adaptive. The ‘trigger’ event model only offers the ability to trigger simple, node level behaviour such as light and sound activation, but is unable to support globally adaptive network behaviour. There are also other factors limiting its widespread adoption, such as the lack of network management, and significant data losses in real-world implementations.

### 2.3.2 SINA

One other notable device database is the SINA system (Sensor Information Networking Architecture) [48]. SINA is similar to other SQL-like device databases, but also translates the idea of database scripting into the sensor-network context. Operations that cannot be performed using traditional queries may be programmed using SCTL (Sensor Querying and Tasking Language), a language that supports scheduled and interactive scripts at both a local and a global level [30].

The availability of a scripting language gives the database greater configurability and adaptability, since it is able to not only collect data, but use the information to optimise resource usage. The addition of these features makes the SINA framework fit into an Active Sensor framework (see Section 2.5).

SINA has shown to be an effective abstraction for networks, while maintaining flexibility. While its similarity to traditional server database systems makes it an easy framework to learn, it does suffer from a few limitations. Firstly, SINA was developed with queries in mind — thus, the scripting language is not as rigorously defined as the querying process. Like server database systems, scripting is more often an administrator feature, with users limited to common query operations. Secondly, SCTL scripts do not support complex mobility rules, relying on manual selection of script targets. Thus, spatial limitations can only be applied to scripts/queries if the user knows what nodes are in what regions. Lastly, there is no means of resource management for running multiple concurrent scripts in the network, making it difficult to support fixed queries and roaming users at the same time.

## 2.4 Communication Abstractions

Communication Abstractions are frameworks that provide facilities for applications to share data and events. They simplify application development by abstracting common features, such as routing and communication, data gathering, and synchronisation.

These abstractions are primarily useful for helping application developers to build robust code and communications, and prevent the re-implementation of common components. Ultimately, the abstractions do not provide enough expressive power for defining the high level behaviour of sensor network nodes. A description of some of these abstractions is featured in Appendix C.

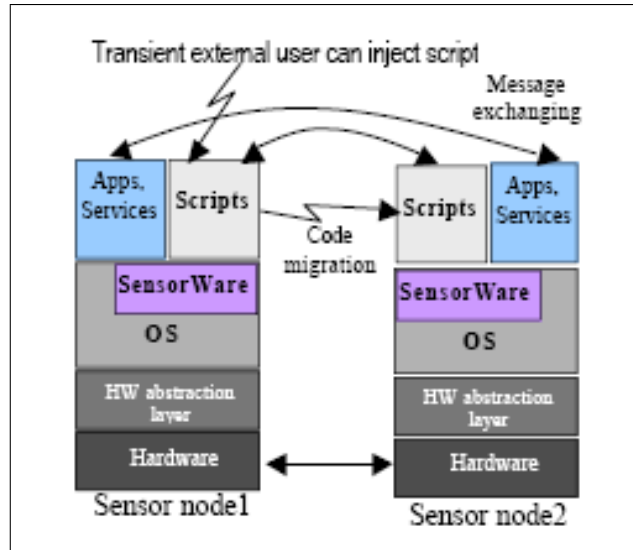


Figure 2.1: Sensorware’s script sharing architecture [10]

## 2.5 Active Sensor Frameworks

### 2.5.1 SensorWare

SensorWare is a WSN approach that “remedies the limited flexibility [of approaches such as TinyDB] at the expense of increased responsibility for the programmer” [10]. It uses migrating control scripts to determine the top level behaviour of the sensor network. Like sensor network databases, there can be multiple active scripts, requesting, transmitting, and managing data in synchrony. This approach allows multiple applications to be shared by the one network, and retains the ability of systems like TinyDB for users to make manual sampling requests (Figure 2.1).

Since flexibility is the highest priority goal for SensorWare, it uses a powerful scripting language based on Tcl [41]. The glue core supports basic expression evaluation and control flow, command and function management, as well as multi-threading and concurrency control. Extensions to this core give SensorWare the complete API for radio-based communication, sensor device interaction, and script migration.

A significant development of SensorWare is the use of complex migration rules, allowing a script to selectively decide how it can propagate throughout the network. This allows the development of fully fledged distributed algorithms,

without compromising the limited bandwidth of the wireless network.

Although several other WSN frameworks are capable of supporting such features, it is notable that SensorWare is one of few that attempts to implement such a fully featured language at an *interpreter* level. It is only capable of running on more powerful platforms than the existing mote architecture — for example palm-top computers and other consumer mobile devices. This prevents SensorWare from running deeply embedded applications, that typically have extreme size, communication and power constraints. However it has demonstrated an effective flexibility/responsibility trade-off, that may become more viable as hardware technology improves [10].

### 2.5.2 Maté

Maté is a virtual machine framework for mote networks, that encapsulates large amounts of complex low level code into high level primitives [34]. Users can insert code into event handlers, called contexts, from both a central server or via roaming users. Contexts consist of sequences of opcodes, with each opcode capturing some essential TinyOS operation or service.

Virtual machines (VMs) are built in an application specific manner, compiling common language level operations (such as arithmetic and control flow) alongside services such as sensor sampling and communication protocols. Maté provides operating system and interpreter-like services to application developers, such as concurrency, memory management and data-typing. This approach has shown to be highly efficient when compared with general purpose virtual machines [49], and offer performance similar to precompiled TinyOS images. Also, the extent that a VM is run-time configurable is up to the VM developer.

This framework is a very effective abstraction, from both application developers, and end-users' perspectives. It provides a solid set of tools and abstractions that make it easy for developers to deploy a safe, efficient and application specific virtual machine — while allowing users to easily configure and adapt the high-level behaviour of the network after it has been deployed.

### 2.5.3 DSWare

DSWare, while not as much of a general purpose abstraction as Maté, is the approach that most resembles ours [37]. Focused on providing a middleware layer, it offers services such as data gathering, communication and synchronization. This approach is similar to SINA, although it supports a more flexible client

scripting language.

The main contribution of DSWare is the inclusion of event-services, offering applications the ability to subscribe to events and event patterns. Using an SQL-like syntax, clients provide sets of events and conditions of which they wish to be notified, and the middleware handles the required gathering and detection process. A unique feature of this system is how it handles uncertainty and unreliability, by providing confidence functions and thresholds. This allows the system to continue detecting event patterns when sensors fail or become unreliable.

One of the current limitations of DSWare is the tight binding between event services, data gathering, and communication — limiting application control of what data is able to trigger events.

The most challenging limitation of this system, is the trade-off between the specification power and its computational complexity. This has been addressed at length by Römer and Mattern, who notes that composite event approaches fail due to the NP-complete time complexity of detection methods [45]. A suggested improvement to DSWare’s approach involves the additional specification of the temporal characteristics for events, and using simple detection algorithms with linear complexity. However, the current specifications supported by both DSWare and Römer and Mattern’s approach have limited applications and domains. This is because they fail to take into account the *ordering* and *cardinality* of events.

## 2.6 Summary

In our brief summary of the approaches and systems above, we have noted that approaches tend towards either a data-centric or control centric approach. Effective, high level abstractions need a good balance of both if they are to be able to handle the wide variety of different applications of wireless sensor networks.

Approaches such as databases fail in this regard — while they have strong data-centric capabilities, support for reactivity and event-handling is limited, and is often only added as an afterthought. Communication abstractions are useful tools for developers, but ultimately are not high-level enough for building complete applications, or to be used by non-programmers. We have noted the active sensor frameworks show the most promise — the Maté virtual machine and Sensorware support both simple scripting for data gathering, and management of complex scripts for adaptive behaviour. However, managing complex event patterns at the script level is a difficult task, especially for non-programmers. DSWare is a middleware system where composite event detection is offered as a service to applications/users, which closely matches our goals. However, there

are limitations in what real-world scenarios can be expressed by these systems  
— a problem we will address in our research.

## CHAPTER 3

# Event Detection

This chapter looks at how active databases have also addressed the problem of detecting composite events, and explores the development of a number of solutions. We go in depth into a recent solution that enables powerful specification of real-world situations, and shows promise for providing specifications in WSN applications.

In working with embedded systems, developers inevitably encounter events of some form, either to meet reactive system requirements, or through event-driven languages. However, the events that systems use are not always singular primitives, as events that occur in the real-world are usually composed of complex patterns. A key problem for developers and end users alike is how to specify and detect these patterns. This is difficult, as it requires bridging the gap between the individual aspects of the world that a WSN node can view through sensors, and the complex relationships of events that make up a real world scenario.

In the previous section, we have explained that there is a need for an effective balance between data-centric and control-centric approaches. The purpose of a sensor network is for retrieving information, but we often need explicit control over how that information is managed and delivered.

DSWare uses an approach that comes close to meeting both these goals, by coupling a data-oriented middleware, with a composite event notification service. The service allows users to specify patterns of events in which they are interested, and either deliver the event data, or provide scripts to enable reactivity. However, the system is limited by the power of the specification. While capable of limited expression of time boundaries (explosion example), applications involving the absence of events (rain), or the counting of events (traffic) are beyond its scope.

Thus, to provide a different perspective, we have looked at how events are handled in *active* database and middleware systems. These systems can give us a new insight in how to create complex behaviour using simple specifications, whilst retaining a balance between data driven and control driven focuses.

## 3.1 Active Databases

Active databases typically function under a normal insert/update/query/delete data model, but give the database the ability to execute actions in response to events, such as altering of data or logging on of a user. Actions range from simple database commands, to the triggering of complete, externally executed scripts, such as notifying authorities or performing analyses.

We will look at how active database technology has developed, from HiPACs simple Event-Condition-Action (ECA) rules [12], to composite event models such as ODE [25] and Snoop [13], and finally to AMIT's powerful situation specification language [5].

## 3.2 Event-Condition-Action

HiPAC pioneered the idea of using Event-Condition-Action rules in databases, in order to efficiently respond to changes in database tables and rows [12]. It consists of sets of rules, each specifying a data event to respond to, a conditional WHERE clause defining what rows and columns the event applies to, and an action to execute when the rule is matched. Actions typically perform additional updates or database administrative functions.

ECA has since been standardised, and are a part of the SQL3 standard, sometimes referred to as database 'triggers'. There is also some support for the Event-Condition-Action paradigm in related systems, such as middleware [52] and expert systems [18]. ECA is the format of database 'triggers' found in WSN database systems such as TinyDB.

The fundamental restriction with the basic ECA paradigm is that it is limited to singular event-action pairs. It is therefore unsuited to finding real-world scenarios that consist of complex composite events. Using ECA rules for these scenarios would require a finite-state-machine model and require many sets of rules, which can be difficult to write and maintain.

## 3.3 Composite Events

In order to support the detection and subsequent reaction to composite events, a number of different models have been proposed. ODE [25] expresses composite events as regular expressions and evaluates them using finite-state-automata; Snoop [13] specifies events as a set of relations and contexts; and SAMOS [21]

uses Petri-nets to evaluate specifications against system state and subject to temporal constraints.

These systems have a similar level of expressiveness as DSWare and Römer and Mattern’s approaches, but in a database environment. They are capable of expressing temporal constraints of an event, such as the ordering of events, or the time in which the events must occur. However, these simple approaches are not suitable for all applications, especially where the temporal context of events is variable. Also, these approaches do not take into account the semantic information reported with events, making it difficult to tailor expressions to react under specific conditions [6]. In addition, these systems have problems with computational complexity, requiring too long to detect events in real time [55], the same problem identified in WSN approaches [45].

Other research models utilise features such as complex event algebra; functional programming; and temporal logic, although these are generally too complex for average user-level database systems. An ideal approach should support powerful event expressions, but remain easy to express simple real-world scenarios, and not require special programming knowledge to understand.

### 3.4 Situation Manager

AMIT (Active Middleware Technology) is a composite event system, and provides services for applications to detect *situations* [5]. A situation is essentially a real-world state, composed from a pattern of composite events and constraints.

Amit uses similar composite event specifications and techniques as ODE and Snoop, but uses several innovations to address the computational complexity of composite event systems, and improves the expressiveness over approaches such as DSWare.

This includes particular attention to the context of a situation, such as temporal context, data associated with events, and the groupings of events [4]. For example, a *lifespan* defines a ‘context of relevance’ for the situation — limiting the detection of the situation to within a temporal span. Not only does this innovation improve the performance of detection algorithms, but allows the inclusion of additional event relations, such as temporal sequences, counting operators and absence operators. Essentially, a lifespan allows us to use a time-sliced, closed world assumption, restricting the solution domain and allowing us to make inferences based on the number of event occurrences.

It is this expressiveness that makes AMIT more suitable for active databases, and also makes it a suitable specification for WSNs. In addition, it uses a mid-

dlaware data-event model similar to DSWare, giving it the balance between data-centric and control-centric approaches, and thus the flexibility for a diverse range of WSN applications.

### 3.4.1 Situation Specification

The main primitive in the situation specification is event types. An event type is application specific, and can encompass things such as sensor readings and system notifications. Each event signaled to the system is an instance of an event type, and provides a set of data based on this type.

Lifespans are the next highest abstraction in the system, and are defined by a *lifespan class*. Each lifespan class is composed of a set of event types, which determine when lifespans are opened and closed. These event types are called *initialisers* and *terminators* respectively.

A situation definition itself is composed of several different components. It must firstly include a *lifespan class*, to define the time intervals where the situation is relevant. Secondly, it must contain a set of event types that contribute to the detection of the situation — these are known as *event operands*. Thirdly, there may be a set of *conditions* defined on the operands that limit what event instances can participate in the situation. These consist of Boolean, logical expressions in conjunctive normal form, and may be declared using the data associated with each event. Lastly and most importantly, a situation must contain an *operator*, which defines the relationship between operands that the situation will detect. The possible operators available in AMIT is shown in Table 3.1.

In addition to the basic definition of a situation, there are a number of policies and quantifiers that control the behaviour of various aspect of the situation. These are outlined in Table 3.2. Also, there is a special class of event types called *internal events*. Instances of internal events are triggered as a result of a successful situation detection. Including internal events in a situation definition allows the definition of *nested situations*.

Further details are available in the AMIT technical report [6].

### 3.4.2 Notation

When specifying situations, a table notation is usually used, which lists each situation attribute and its associated value. This matches the XML schema that AMIT expects when defining situations. To better visualise multiple situations, we will instead use a diagrammatic notation, shown in Figure 3.1.

Class	Operator	Description
Joining	$all(E_1, E_2, \dots, E_k)$	a conjunction of events $E_1 \dots E_k$ with no order importance
	$sequence(E_1, E_2, \dots, E_k)$	an ordered conjunction of events $E_1 \dots E_k$ where event $E_i$ precedes event $E_{i+1}$
Counting	Counting operators designate a conjunction of $n$ events, weighted according to each operand. A situation with a counting operator is triggered when the total weight of collected events satisfies the operator.	
	$atleast(n, E_1, E_2, \dots, E_k)$	a minimal conjunction of $m$ events out of $E_1 \dots E_k$ such that the total weight of the $m$ events is more than $n$
	$atmost(n, E_1, E_2, \dots, E_k)$	a maximal conjunction of $m$ events out of $E_1 \dots E_k$ such that the total weight of the $m$ events is less than $n$ within the lifespan. Must be detected at the end of the lifespan (i.e., deferred detection mode).
	$nth(n, E_1, E_2, \dots, E_k)$	a conjunction of $m$ events out of $E_1 \dots E_k$ such that the total weight of the $m$ events is exactly $n$ .
Absence	Situations with an absence operator are always detected at the end of the lifespan (i.e., deferred detection mode).	
	$not(E_1, E_2, \dots, E_k)$	none of the events $E_1 \dots E_k$ has occurred within the lifespan.
	$unless(E_1, E_2)$	occurrence of the first operand and the nonoccurrence of the second within the lifespan
Temporal	Situations with a temporal operator are always triggered when they occur (i.e., immediate detection mode).	
	$every(t)$	a period of $i * t$ time units since the initiation of the situations lifespan, where $i > 0$ .
	$after(E, t, p)$	a period of $t$ time units since the occurrence of $E$ . Policy $p$ determines the policy when two instances of $E$ occur in less than $t$ . There are three possible policies: <i>Add</i> both occurrences of $E$ are considered. <i>Ignore</i> the first occurrence of $E$ is considered and the second occurrence is ignored. <i>Replace</i> the first occurrence of $E$ is ignored and the second occurrence is considered

Table 3.1: Definitions of different situation operators [6]

Policy	Affects	Modes	Description
Duplication policy	Lifespan initialiser	<i>add, ignore</i>	Determines what initialiser events trigger new lifespans. Policy <i>add</i> always adds a new lifespan; <i>ignore</i> ignores events if there is already a lifespan open.
Detection policy	Situation detection	<i>immediate, deferred</i>	Determines when detection is triggered. <i>Immediate</i> will trigger detection upon receiving a valid operand; <i>deferred</i> only runs detection once a lifespan is terminated.
Termination quantifier	Lifespan terminator	<i>first, last, all</i>	Determines which open lifespans are terminated. <i>First</i> terminates the oldest lifespan; <i>last</i> terminates the newest lifespan; and <i>each</i> terminates all open lifespans.
Expiration policy	Lifespan termination	<i>timed, event-only</i>	Optionally allows lifespans to close on expiration of a timer. <i>Timer</i> allows a (fixed) timer to trigger termination; <i>event-only</i> only causes termination via terminator events.
Consumption quantifier	Operand consumption	<i>first, last, all</i>	Determines which events are consumed in a situation. <i>First</i> consumes the oldest valid event(s); <i>last</i> consumes the newest valid event(s); and <i>all</i> triggers all valid events.

Table 3.2: Definitions of situation quantifiers and policies

### 3.4.3 Situation Example

To make this specification more concrete, we shall work through a simple example.

Let us consider the problem of detecting when there is a risk of plant dehydration due to a lack of soil moisture. Suppose the main requirements is to find when the soil is dry and it has not rained for a week, and the soil is still drying at a rapid rate. Using regular soil moisture sensor samples, and events triggered from a rain sensor, we can specify suitable situations that will detect these circumstances (see Figure 3.2).

There are several key ideas behind our chosen way of specifying these requirements, which demonstrate the power of AMIT’s situation expressions. Firstly, by breaking the requirements into two distinct situations, we are able to receive two separate event notifications — a *no-rain* event, and a *plant-at-risk* event. We

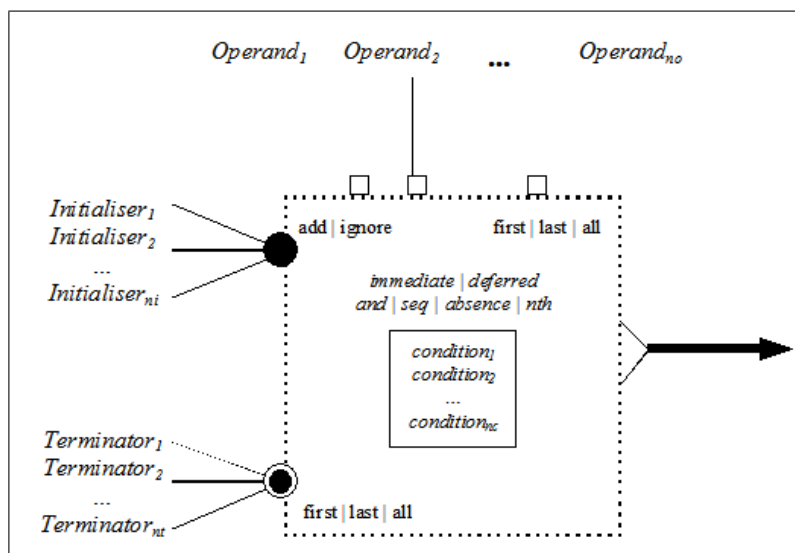


Figure 3.1: Diagram notation for situations. Outside of the situation, the top-left port is connected to incoming initialiser events; the bottom-left port is connected to incoming terminator events; and the top-central port(s) are connected to incoming operand events. Inside the situation are the policies for each port — top-left is the duplication policy; bottom-left is the termination quantifier; and top-right is the consumption quantifier. The centre of the situation, from top to bottom, contains the detection mode; the situation operator; and a box containing a list of conditions. The triggered internal event is fired on the arrow leaving the right-port, and may be connected to an incoming port of other situation(s). This notation is therefore especially useful for visualising situation nesting.

may decide to take some action after an occurrence of the *no-rain* event, such as turning on sprinklers or adjusting the soil-moisture sampling rate. Secondly, using policies such as the *add* duplication policy gives the user additional control over when time intervals apply — in this case, it lets us concurrently keep track of multiple *rain* timers with respect to soil moisture readings. Thirdly, we have used the *sequence* operator to find significant drops between successive soil moisture readings. Lastly, we have used the *not* operator, with a *deferred* detection policy, in order to find periods when rain has *not* occurred. This is a unique feature that can only be expressed using the situation definition approach of AMIT.

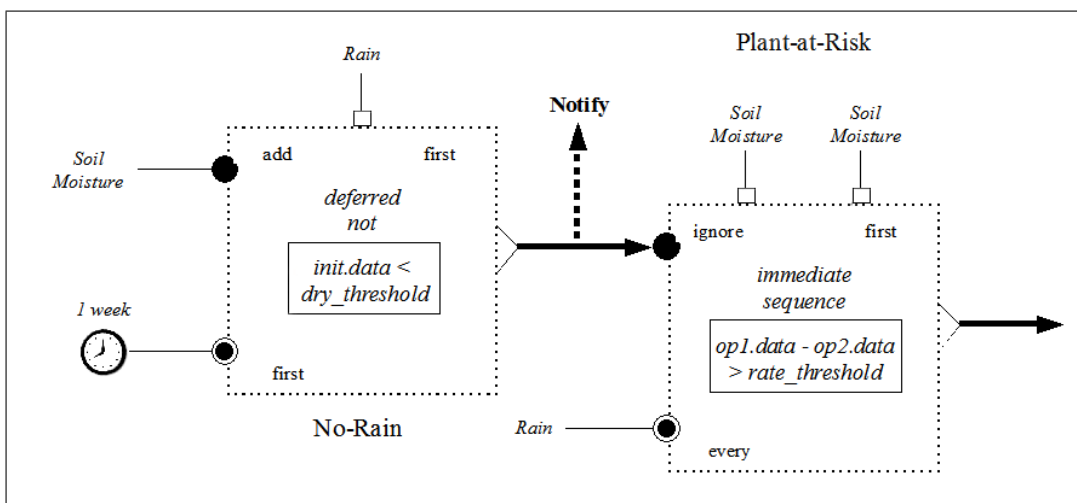


Figure 3.2: Example situation: Plant dehydration risk

### 3.4.4 AMIT Architecture and Operation

Figure 3.3 shows the general architecture of the AMIT system [40]. The responsibilities of the different components of situation detection process is as follows:

The Definition Manager (DM) contains situation management parsers, for checking situations for errors, loading situations into the system, and notifying the Event-Query-Processor (EQP) that there are new situations to detect. The Input Dispatcher is responsible for taking formatted events, storing them according to the situation definitions, and triggering the EQP. EQP constructs the relevant lifespan views over the event database, builds and executes queries for the defined situations, and applies any additional logic as necessary. Detected situations are then passed to the Input Dispatcher for further processing, including client notification, internal event processing, and event consumption.

The complete details of how AMIT performs all these steps is beyond the scope of this paper, although we may refer to aspects of AMIT’s approach, and will endeavour to explain any background as required. Interested readers should read the full AMIT technical report [6].

### 3.5 Our Contribution

With the specification of situations in mind, we make our goals more concrete. We aim to port the semantics and execution environment of AMIT into a form suitable for running on the memory, speed and power constrained devices of sensor networks. We then evaluate the expressiveness and performance of the system, in order to measure its feasibility as a tool for building situation-aware applications. We lastly determine what additional enhancements to the specification and performance are required to improve its suitability for sensor networks. The system will be simply be referred to as SENSID, or a Sensor Network Situation Detector.

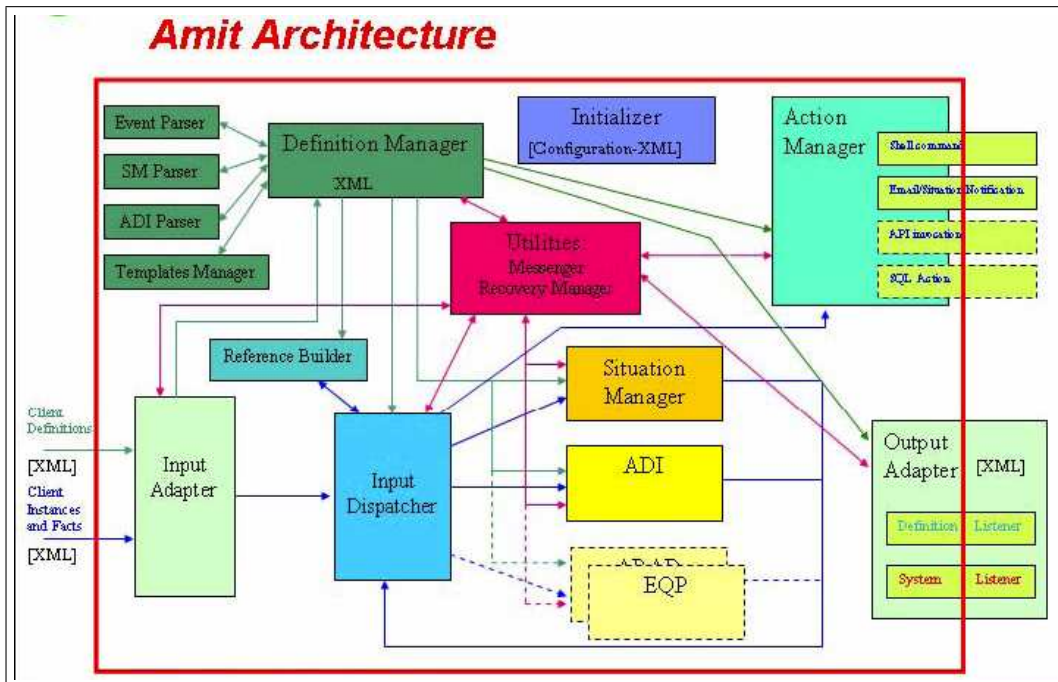


Figure 3.3: The AMIT Architecture [40]

## 3.6 Project Scope

It is *not* the focus of this project to create a fully-fledged middleware system, but to examine the feasibility of using such a tool to aid development. Therefore, while we may use WSN networking to load definitions and deliver notifications, we will only consider single node situation detection, using events produced by local sensor samples. In particular, we will not be looking at how wireless sensor networks can perform detection within spatial contexts.

It is hoped that the specification can be further expanded to support the unique features of wireless ad-hoc sensor networks. There is already investigation of distributed situation detection in recent work, including parallel detection [3], and spatial context awareness [4, 44].

## CHAPTER 4

# SENSID Design

Our primary goal is to demonstrate that a sensor network situation detector is a feasible concept, and as such we have designed and implemented the prototype system SENSID. This chapter looks at the design of the system, examining the operation on a component level, and taking note of what challenges needed to be addressed during implementation.

The basic design of SENSID reflects the pattern of execution of AMIT, briefly described in Section 3.4.4. In order to detect situations such as the *plant-at-risk* example, the following steps are required:

**Situation Subscription** : Handles the process of registering definitions and notifying the appropriate sub-systems.

**Event Dispatching** : Dispatches events to various subsystems, depending on how the event is defined in situations.

**Lifespan Management** : Opens and closes lifespans subject to the situation's lifespan class.

**Operand Filtering and Storage** : Stores event operands according to situations, ready to be accessed for situation detection.

**Situation Detection** : Runs detection algorithms against stored events to determine if a situation has occurred.

**Situation Notification** : Fires internal events to allow nested situations to occur, notifies clients of situation occurrences, and performs event consumption.

We have modelled these steps using a component based architecture, featured in Figure 4.1. More about how these components are translated to a TinyOS implementation is discussed in Chapter 6.

The remainder of this chapter is concerned with these components and the steps they represent.

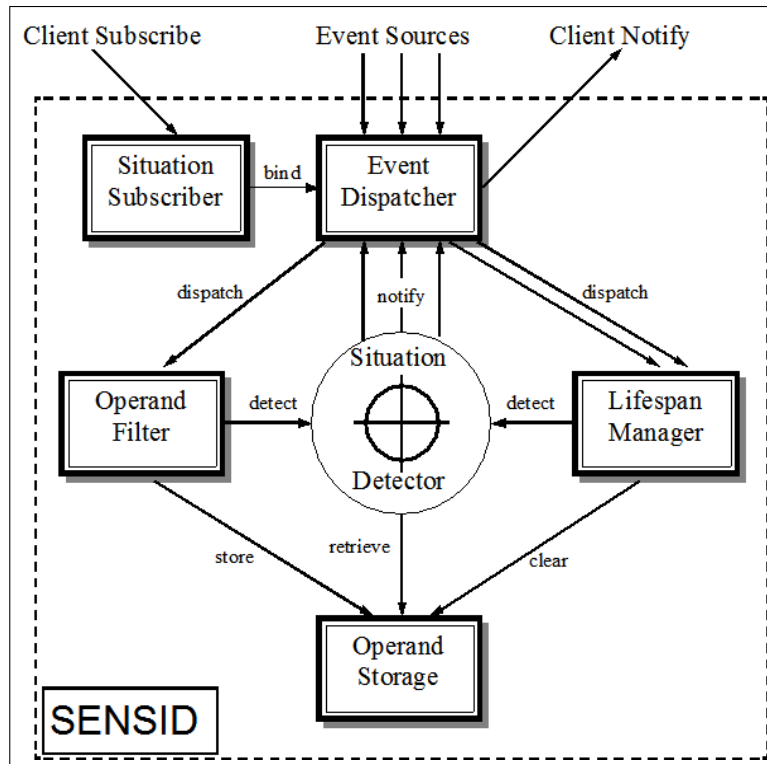


Figure 4.1: SENSID component diagram. Boxes represent system components, arrows represent flow of control, and the dotted line shows the system boundary.

## 4.1 Situation Subscription

The process of subscribing to a situation of interest is a critical part in the design of the system. The situation definition is required for the system to be able to effectively store and manage incoming events.

There are several pieces of information that a client application must provide, including:

- The *event-types* that contribute to the situation,
- A set of *conditions* that event instances must pass to be valid,
- What event relationship will trigger a match (*operands*),
- A *lifespan* during which the situation is relevant, and
- *Quantifiers* and *policies* to affect behaviour of lifespan, detection timing, and event consumption.

We have adopted a publish/subscribe model, a method often used in database systems and middleware. In this model, the server publishes a set of event types that of which it is aware, by making the information available to clients. One or more client applications subscribe to these events, so they can receive notification when they occur. In our system, clients will instead subscribe to *situations*, rather than the individual events.

Instead of publish/subscribe, we could have opted for a query based approach where presenting a situation definition and receiving occurrences is a single operation. However, we chose a publish/subscribe approach since it allows clients to respond to situations as they occur, and conserves memory by filtering unnecessary events — two advantages suitable for a constrained, embedded application.

Each subscription is uniquely identified by the system, allowing clients to pause or cancel them at any time. Since clients may subscribe to multiple situations at once, situation nesting is also possible.

Our implementation must find a way to succinctly represent all details of the specification, while still supporting a wide range of situation definitions.

## 4.2 Event Dispatch

Once one or more clients have subscribed to situations, the system uses the definitions to trigger the rest of the detection process. To simplify the management of incoming events, we use a Dispatcher design pattern [23, 20]. In this approach, all events are signaled on the dispatcher component, which forwards each event to one of several handling components.

The advantage of this approach is fourfold. Firstly, event producers only need to signal events on the dispatcher itself — they need no prior knowledge of other components in the system. Secondly, a dispatcher’s behaviour can be configured subject to local state information. This is suitable for our system, where the dispatcher may not know where to forward events until it has received situation definitions. Thirdly, the dispatcher can cache received events, allowing the event-producer to resume immediately, while the dispatching occurs in lower priority tasks. Again, this is desirable for multi-threaded embedded systems, where critical components should not spend a long time performing event signaling. Lastly, the dispatcher can immediately add additional meta-data and system specific information to the incoming events, such as time-stamping and event-typing — steps which are cumbersome to perform in higher level components.

We accomplish event dispatching using event-type binding. Given a situation definition, the dispatcher must record what events will affect lifespans, which

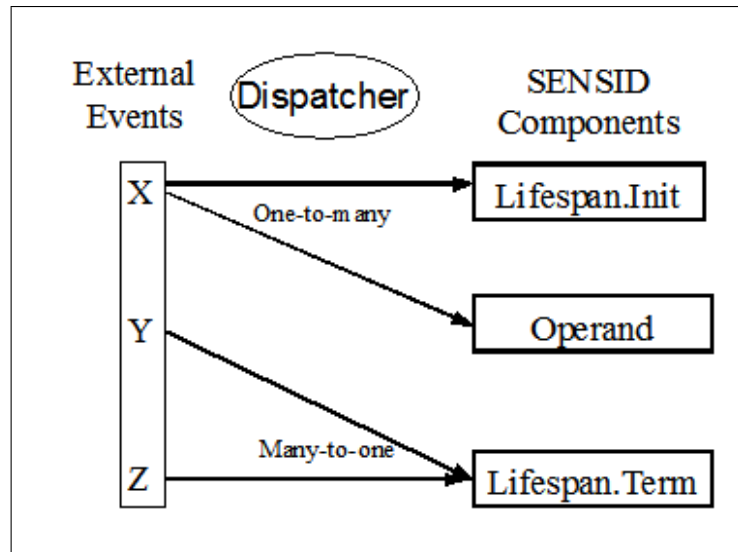


Figure 4.2: Example dispatcher configuration, showing the flexible manner in which events can be bound. Event X has a *one-to-many binding*, since it is bound to both the lifespan initialiser and operand ports. Events Y and Z have *many-to-one bindings*, since they are both bound to the lifespan terminator port.

events will be considered operands, and which events must be reported to the client. This information is called bindings, since it effectively binds each event type from the dispatcher to a set of ports on other components. Bindings allow us to achieve varied and flexible dispatcher configurations, such as in Figure 4.2.

### 4.3 Lifespan Management

Lifespans represent a temporal context of relevance for situations, whereby events of interest are only recorded during the period a lifespan is open. This helps us model real-world scenarios, and aids computation and efficiency by limiting the search space in which the detection algorithm must run.

When an event is signaled that is bound to a lifespan class definition, it is dispatched to the Lifespan Manager component for handling. This component is responsible for opening, closing, and keeping track of all lifespans in the system.

The Lifespan Manager receives events from the dispatcher through two ports — the initialisation port, and the termination port. The behaviour of these event handlers varies depending on the policies and quantifiers defined for the lifespan.

We support the same set defined by AMIT, as described in Table 3.2

Note the relation between situations and lifespans is one to many — a situation may have many lifespans, yet a lifespan is only active for a single situation. Thus, an event that fires a termination signal on the lifespan manager will only terminate lifespans for a single situation.

The Lifespan Manager has two additional responsibilities during lifespan termination. Firstly, it is responsible for triggering a detection process for the lifespan's situation — this is called *deferred detection mode*, and will only occur if explicitly set in the situation definition. Secondly, it is responsible for performing garbage collection for the situation. This involves the removal of all significant events that occurred during the lifespan, as long as they are not valid for other situations. This is an important step, as rapidly freeing unused memory is a critical requirement of memory constrained WSN nodes.

## 4.4 Operand Filter

We refer to events instances in different ways, depending on what point that instance is being handled by the system. Below is a short description of the different terminology:

**RawEvent** : Entered system through the dispatcher's incoming port. Enters a port depending on the event type. Contains a small amount of associated data.

**BoundEvent** : Event has been dispatched to a sub-component. Metadata such as timestamps has been added.

**Operand** : The event is marked as an operand for a situation. It has passed basic conditions on the event group, and is defined by at least one situation that has an open lifespan.

**Candidate** : An operand that has been selected to match a situation.

**Participant** : A candidate that has passed all conditions in a successful situation match.

Along the process from *RawEvent* to *Participant*, it is important to determine what events will become operands for situations. To find valid events, and to filter out events that are not, we employ the use of an *Operand Filter* component. This component receives events directly from the event dispatcher, runs a series of

checks against each event, and forwards valid operands to the next component in the system.

The first basic check that must be performed is a test against lifespans. Since the purpose of lifespans is to provide a context of relevance, we must ensure events occurring outside of lifespans are not forwarded by the system. This can be done by checking if a lifespan is open for each situation in which the event-type is defined. If no lifespans are open, the event is simply discarded.

At this time, in addition to the lifespan check, the system may evaluate singular conditions. Singular conditions are those that involve only a single event — for example threshold tests such as  $Event.data \geq Constant$ . If the event fails to pass all singular conditions on all situations, then it is discarded. If there are situations that do not define any conditions for an event-type, then it will unconditionally pass this test.

Note that even if an event has failed several filters, it only needs to pass for a single situation for it to be stored. While this may lead to later reprocessing the filters, delaying the filtering process until the detection phase would be significantly more wasteful. We believe that performing all possible checks as early as possible is essential to minimising memory usage, and subsequently minimising processing time through a smaller search space.

Once an event passes all filters, it is forwarded to a storage component for later access. The only other responsibility of the Operand Filter component is the triggering of detection for situations that are in *immediate detection mode* (Section 4.6).

## 4.5 Operand Storage

In order to provide efficient and effective access to potentially large numbers of events, we have decided to use a dedicated event storage component. It is responsible for the soft-storage and management of all operands received from the filter, and provides interfaces for retrieving and removing stored operands. Since we are limited by severe memory constraints, the design of storage and retrieval algorithms is an important part of this system. In the AMIT framework, there are no such limitations, since the backend for storage is a file-system or DBMS, capable of managing and accessing large number of events.

We use the notion of a simplified database *query* to provide an interface to the event retrieval algorithms. Using a query structure enables us to specify properties of the events to be retrieved, such as data ranges and retrieval ordering.

This allows us to support different detection strategies, while keeping the retrieval process close to the data so as to maximise efficiency.

The functionality may also be extended independent of the actual query interface, enabling future features such as hard-storage, compression and multi-threading. There are currently commercial embedded database systems such as DB\* that would give a substantially more robust and expandable back-end, however the memory requirements for these systems are still too high for the extreme constraints of motes.

This component also provides the ability to remove events from the storage, although determining *when* to remove events is the responsibility of other components, such as the garbage collector and situation detector. The actual removal process is quick and efficient, since the memory may soon need to be recycled for new incoming events.

## 4.6 Situation Detection

Once events have been signaled, dispatched, filtered and stored, the system must undergo a detection phase in order to determine if a situation has been satisfied. Since there are several different ways in which detection is triggered, (see *detection policy* in Table 3.2), we have dedicated a separate system component for managing this process.

We call this the *Situation Detector*, and it is responsible for accepting detection requests, evaluating the current state of the operand store, and reporting valid situation occurrences. The algorithms used in this process must be lightweight and fast in order to satisfy the hardware constraints, while remaining correct and complete. Ideally, it should be able to completely evaluate each situation within the time between two events — while this is environment dependent, it would allow evaluation to occur in *immediate detection mode*, without imposing extra latency.

However, since we cannot guarantee the rate at which events enter the system, it is preferable to relax the speed constraints in favour of ensuring that detection will *eventually* occur. In order to do this, we need to address a couple of problems. Firstly, we need to ensure that all signaled detections will occur, even when some detection processes require a long time to finish. Secondly, we need a way of limiting memory usage, as the system cannot handle scenarios where the rate of event input is greater than the rate of event handling.

We have addressed the first problem by queuing each detection request, and performing each detection algorithm in a single, low priority thread. We allow

the detection process to be preempted by incoming events, and thus maintain the throughput of the system. The second issue is resolved by allowing only one detection request per situation to be queued at a time. Then, to ensure that the correct number of situations are detected, the number of requests is counted and taken into account when the detection algorithm runs.

When the detection process actually runs, queries are used to selectively retrieve events from storage and evaluate if the specified situation has occurred. In AMIT, a backtracking algorithm is used to repeatedly select and evaluate operands and conditions, until the situation is matched, or there are no more instances. Evaluating the performance of this algorithm, and determining if there is a more suitable algorithm for sensor nodes is part of our implementation in Chapter 5.

We currently focus on operators such as *all*, *not*, and *sequence*, as we have found them the most useful for many systems. However, other operators from Table 3.1 are straightforward to implement within the current design.

## 4.7 Situation Notification

When a situation has been successfully detected, it is important that the subscriber is notified so that actions can occur in a timely manner. SENSID, like AMIT, is middleware, and as such it does not directly provide mechanisms for actions. Instead, it is only a matter of ensuring that the appropriate events fire, allowing the subscribing applications to perform actions. It is important, however, to give access to participating event instances so that the subscriber can react accordingly.

Other considerations in this stage include firing internal events to trigger nested situations and consuming participating events subject to the *consumption policy*.

## CHAPTER 5

# Results: Implementation

## 5.1 Memory Management

One of the most important decisions in implementing SENSID is deciding how memory is allocated and managed. Since the system operates as an intermediary between the event-sources and applications, it is difficult to foresee how much memory should be allocated for various data structures. We need to ask questions such as “how big and how many situation definitions will there be?”; “will there be multiple lifespans?”; and “what is the average rate of event production and consumption?”. It is therefore logical to use a dynamic approach, where memory is allocated and freed as necessary. This involves the use of a dynamic memory allocator, such as the *malloc* function in C.

There are a number of issues using a traditional memory allocator like *malloc* on constrained devices. Firstly, there is no memory protection, making it easy to overwrite application memory and cause errors that are difficult to diagnose [2]. Secondly, the event-driven execution of embedded systems makes it difficult to track allocation and ensure all memory is correctly released.

There are memory allocators suitable for node devices, such as *TinyAlloc*, developed for *TinyOS*. It resolves the issue of memory protection, however it still requires a significant processor overhead — and is therefore unsuited for a series of small allocation/deallocations.

We have instead chosen a hybrid static-dynamic approach, whereby pools of memory are reserved at compile time, and may be efficiently and safely allocated at run time. In this approach, we physically store typed data within homogeneous arrays and build dynamic structures through the use of memory pointers.

This is done by firstly encapsulating dynamic data types within memory nodes, called frames. Each frame consists of the typed data, and a memory pointer linking to another frame of the same type. At compile-time, we set aside array(s) of space for frames, allowing us to tune sizes specific to the mote envi-

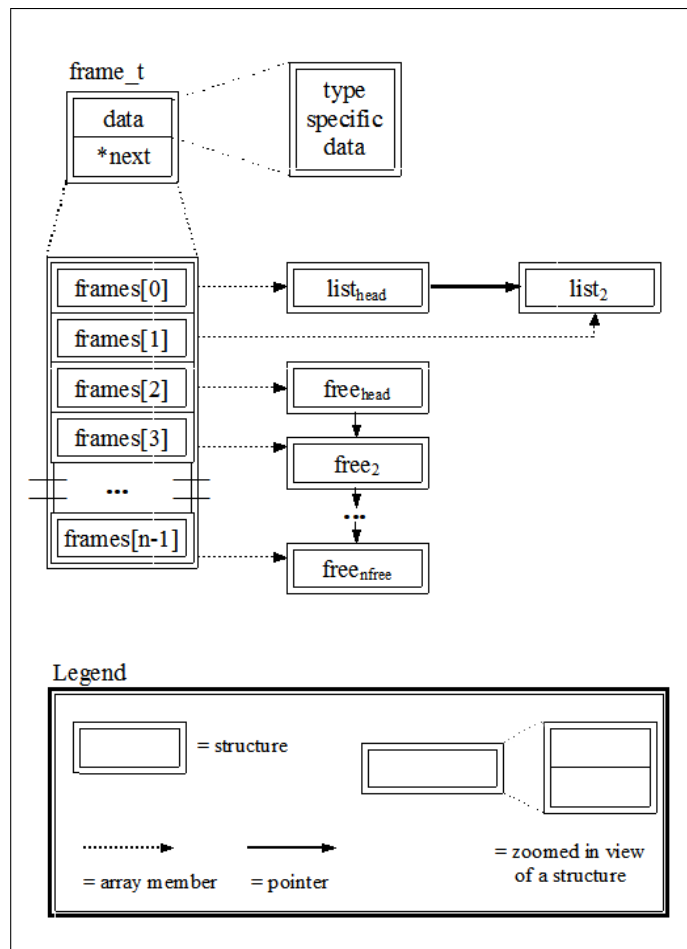


Figure 5.1: An internal representation of memory management, along with a simple data structure.

ronment. At run-time, the frame pointers are arranged to form a stack of free space, giving us a simple, constant time allocation and deallocation.

When the frame is in use (allocated), it is not a part of the *free memory* stack, and thus the frame pointer is unused. We can therefore reuse the pointer for building data-structures, such as lists and queues.

The major difference between this approach and other memory stack techniques, is that we use typed data and typed memory frames. This prevents fragmentation and alleviates the need for memory compacting, since all frames use the same amount of space. Another advantage is that we can tune the memory sizes specific to the type of data stored — for example, allowing a large

amount of space for events, but only a minimal amount for situation definitions. The disadvantage of this approach is that it is difficult to create generalised allocation/deallocation functions and still preserve data typing, while using type-specific functions introduces redundancy and increases the size of the code base.

## 5.2 Component Specific Data Structures and Algorithms

### 5.2.1 Situation Definition/Subscription

A Situation Definition consists of different types of data — some are required, others are optional; some are fixed size, others are variable. To accommodate these different layouts, we use a three stage subscription process. Firstly, the subscriber provides required data, including the detection operator and detection mode (Tables 3.1 and 3.2). The system may then allocate space for fixed size meta-data, and fill out the required fields. Secondly, the subscriber can provide optional and variable size information, including operands, lifespan initialisation and termination events, and condition sets. The system allocates the required space and builds the necessary lists to hold the situation. Lastly, the subscriber signals that the situation is complete, allowing the system to assign default values for unfilled variables and load the situation bindings. The subscriber receives a unique reference to the situation, both for comparing different situations, and for later unsubscribing.

The three types of variable-size data we need for situation definition is *operand events*, *lifespan events*, and *conditions*. These consist of a set of data nodes, encapsulated in memory frames and stored using the memory management technique outlined in the previous section.

#### Situation Definition Nodes

To store operands, lifespan initialisation and lifespan termination types in a uniform manner we use a list of Situation Definition Nodes (SDNs). Each node contains the parameter this will affect (init, term or operand), the event type that causes it, and any special quantifiers or policies. By drawing on a pool of statically allocated nodes/frames, we can allocate an indefinite number of lifespan events and operand events to a situation, without having a fixed upper limit and without wasting space. We can also reuse the information in forming the event-bindings (see Section 5.2.2).

## Condition Nodes

To store conditions in an efficient, uniform manner, we use a similar notation to conjunctive normal form. The entire condition expression consists of a set of conjuncts, each consisting of a simple logical expression.

There is a considerable tradeoff between representational and computational complexity, the memory requirements, and the flexibility of the logical expressions the system supports. We have adopted a simple representation, favouring short, fast evaluations over a more advanced expressiveness. The conditions the system supports is of the form:

$$x.f \pm xc \text{ } OP \text{ } y.f \pm yc$$

where  $x$  and  $y$  are event instances of either the operands, initialisation or termination events for the situation;  $f$  is the field to compare, either the event *data*, or *timestamp*; and  $OP$  is one of the logical operators  $\leq, <, =, \neq, >, \geq$ .

This can be internally represented using the form:

$$x - y \text{ } OP \text{ } c$$

which is succinct, and easy to compute. With this format, we can represent singular ‘threshold’ conditions (comparison against a constant, where  $y = 0$ ), simple 2-variable comparisons, and offset calculations. Since it is sometimes useful in sensor networks to recognise the *magnitude* of difference between readings, rather than the direction of difference, so we also offer the formula in an absolute value form:

$$|x - y| \text{ } OP \text{ } c$$

which take very little extra to store or evaluate. One point to note about this representation is that it cannot handle conditions involving multiples — it is limited to additive and subtractive arithmetic<sup>1</sup>. It is, however, suitable for most simple calculations, and is economical in size: requiring just a few bytes to store each conjunct.

---

<sup>1</sup>We could further expand this using the form:  $|x \text{ } ARITH \text{ } y| \text{ } OP \text{ } c$  where *ARITH* is either the operator minus  $-$ , or divide  $/$ . This would give us the flexibility of comparing either the offset between two values, or the relative scale of two values — although the representation cannot support both at one time.

## 5.2.2 Dispatcher

### Bindings

In order for the dispatcher to build a set of bindings, it needs to have access to SDNs for each situation. The problem is a matter of performance — to bind an event-type to a component-port requires going through every node of every situation definition, and comparing the event types.

One way we could manage this is by storing the event nodes with respect to the event-type. Each binding would be stored in a list, with each list indexed by an event-identifier — making dispatching a simple matter of iterating over a list. The lists could be built using the memory management system from section 5.1. The only problem with this approach is redundant storage — although we have improved access times, we are also storing each node in both the dispatcher, and the situation definition.

An easier method involves embedding additional pointers in the memory frame for each SDN. Normally, we use a single pointer in each node to link either the free-memory stack, or a situation definition list. By adding another pointer, we can also allow the same event node to be a member of a binding/dispatching list.

In this manner, we only store the information once, but are able to access it in different manners. The only additional overhead is the cost of memory pointers in each node, and pointers for the head of each list. Since event producers are typically connected to the dispatcher at compile time, we can use a fixed size array of pointers to store the heads. An example of this combined structure is shown in Figure 5.2.

### Internal Events

Internal events are those events that are triggered by successful situation detections. In order to allow nested situations, we must have a way of binding and handling internal events. We have chosen to distinguish between external and internal event types using a simple bit-masking approach, allowing us to differentiate between them for the purposes of dispatching, but use the same components for handling lifespans, filtering, storing and detection.

The main change needed to support internal events is to maintain two separate bindings lists — one for internal, and one for external. While the number of external bindings lists is fixed at compile time, the number of the internal bindings lists is dependent on the number of subscribed situations in the system. It is best,

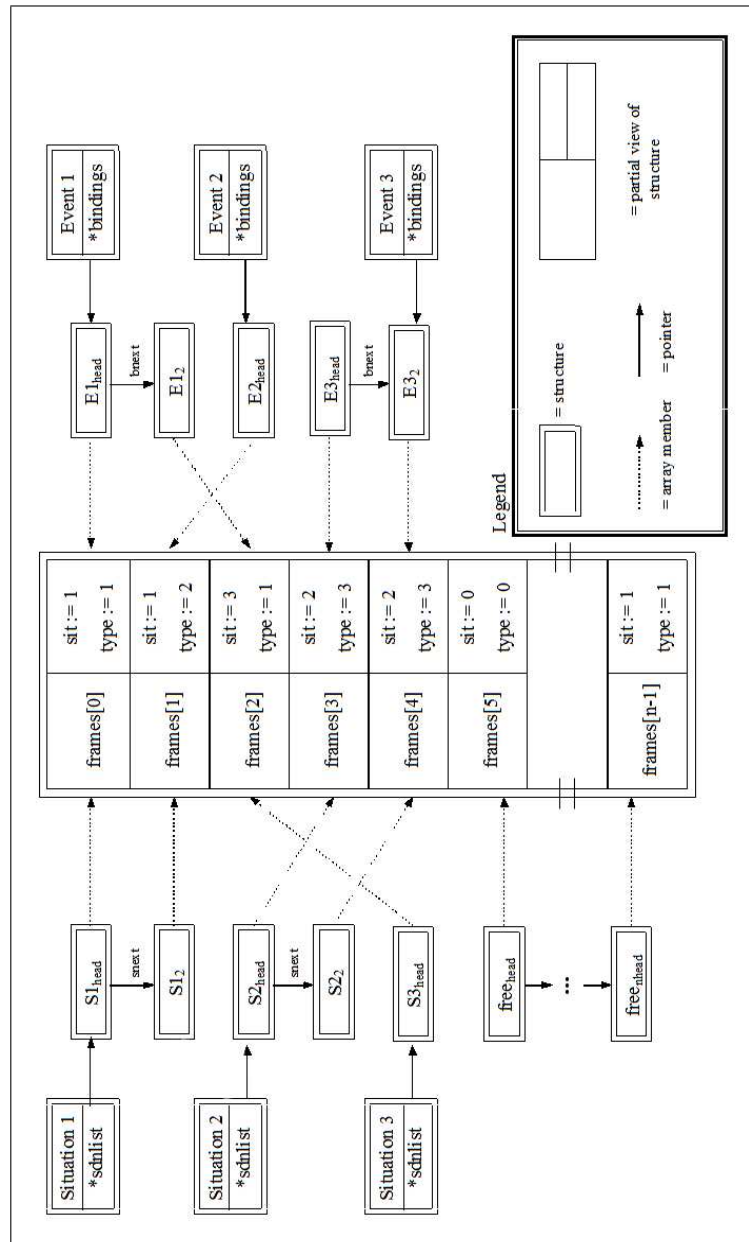


Figure 5.2: An internal representation of the data structure, combining situation definition lists with dispatcher binding lists. The array in the centre is the actual static store for each data frame. The nodes on the left shows how the frames are viewed by situations, in terms of lists of *situation definition nodes*. The nodes on the right shows how the frames are viewed by the dispatcher, in terms of lists of *bindings*, sorted by *event type*. There is also a stack keeping track of free memory frames.

```

EVENT-DISPATCH(e,p)
1  ▷ Dispatches event e from port p
2  TIMESTAMP(e)
3  if ISINTERNAL(p)
4      then SETINTERNAL(e)
5          list ← internalBindings
6      else list ← externalBindings
7  sdn ← HEAD(list)
8  while sdn ≠ NULL
9      do if sdn.type = INIT
10         then FORWARD(e,LIFESPANINIT)
11         elseif sdn.type = OPERAND
12             then FORWARD(e,OPFILTER)
13         elseif sdn.type = TERM
14             then FORWARD(e,LIFESPANTERM)
15         sdn ← sdn.bnext

```

Figure 5.3: The event dispatching algorithm

however, to provide enough space for the maximum number of situations that may be subscribed. While space may be wasted when few situations are defined, most applications have a low limit on the number of situations, and since only pointers for each list are stored, this amount is of little consequence.

#### Dispatching algorithm

The actual dispatching algorithm itself is a straightforward process. It must firstly timestamp incoming events, so that situations involving temporal conditions can later be satisfied. It then finds the correct bindings list, using the event type, and whether the event is internal or external. It is then a process of traversing the list, dispatching the event to the appropriate component based on the port bindings. Pseudocode for this algorithm is shown in Figure 5.3.

One critical aspect of the dispatching process is the order that events are dispatched. We need to ensure that initialisation event forwarding occurs before operand forwarding, which in turn occurs before terminator forwarding. If this guarantee is not met, and an event is used as both an operand and a lifespan controller, then the event will be unable to participate in the lifespan(s) it affects.

We currently ensure this ordering by sorting bindings by port-type during the subscription process.

### 5.2.3 Lifespan Manager

The *Lifespan Manager's* primary responsibility is to add and remove lifespans. Each lifespan data type includes properties for the lifespan's situation, and holds the events that led to its initialisation and termination.

In order to efficiently store and access lifespans, and support the different policies and quantifiers, we have devised a multi-purpose list structure.

It is preferable to access each lifespan with respect to their parent situation, so we will use an array of lists of lifespans. Thus, the addition of each lifespan is performed in constant time through array indexing and list pointer assignment. Termination, however, is more difficult, since we need to support the quantifiers *first*, *last* and *each*. We have chosen to add additional pointers to the lifespan memory frames, to allow the construction of a doubly linked list. We can then access each list as both a FIFO queue, and a FILO stack, which is able to perform *first* and *last* termination in constant time. Supporting the *each* quantifier still requires a complete list traversal regardless of the access method used.

One of the more challenging aspects of implementing the lifespan manager was determining how to perform event garbage collection. We assume that the lifespan manager has access to the event store, and is able to delete selective events based on time and type.

Our initial implementation of a garbage collector simply tracks the situations that include an event type, and the number of open lifespans that are associated with them. If there are no lifespans open for a particular event type, then there cannot be a situation match for those events instances, and can therefore be removed from storage. Figure 5.4 shows an example of the garbage collection process. This is a simple strategy that can effectively reclaim memory in many circumstances. It does suffer some limitations, however, such as when a situation never closes all of its lifespans, then garbage collection will not occur. Similarly, if an event type shared between two or more situations has *interleaved* or *overlapping* lifespans, then garbage collection never runs.

A better garbage collection strategy would take into account the time that lifespans are opened and operands occur. On a lifespan termination, the system could examine the next oldest lifespan, and find any events that occurred prior to its initialisation. If these events are not associated with any other situation, or if the lifespans of all other situations are *newer*, they may be reclaimed.

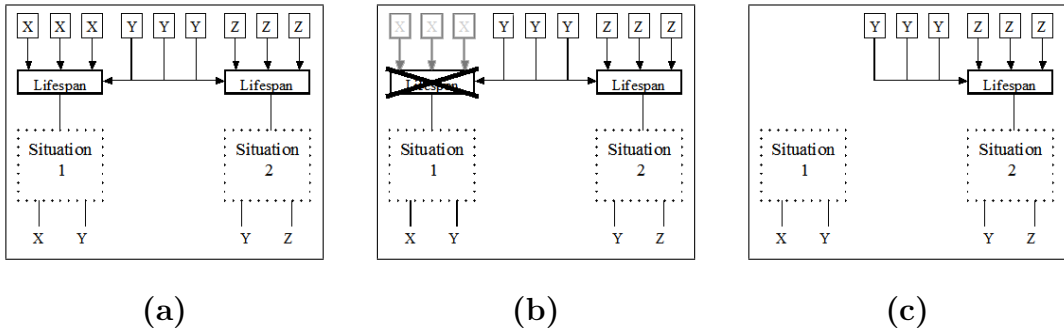


Figure 5.4: Demonstration of simple garbage collection, showing the event-lifespan dependencies **a** before, **b** during, and **c** after garbage collection occurs.

This strategy is able to cope with environments where few situations are detected in the presence of interleaved lifespans. Rather than accumulating unnecessary events, some will be garbage collected on each lifespan termination. Unfortunately, there are some issues that need to be resolved before this is viable, such as how to search for lifespans between different situations. Also, the garbage collection technique needs to be configured with respect to the quantifiers and policies for each situation. The analysis and implementation of more sophisticated garbage collection strategies is left as a subject of future work.

## 5.2.4 Storage

Before we describe how events are stored, we shall firstly define the event query process.

### Query Format

We do not need fully-fledged queries in the same sense as conventional databases, but we would like to select event instances (rows), subject to constraints on event properties (columns). To reduce the computational complexity of performing a query, we use a simple bounding search — where we specify a range of events to retrieve. A wildcard (\*) operand is also available, to indicate an unspecified restriction on upper/lower bounds, and search order reversal is allowed.

For example, to select all events of type  $e$  that has occurred after time  $t$ , we would use the query where type has bounds  $(e,e)$ , time has bounds  $(t,*)$ , and data has bounds  $(*,*)$ .

## Format and Algorithm

To efficiently access events in this manner, we use an array of lists — indexed by event-type (like bindings, section 5.2.2), and doubly-linked (like lifespans, section 5.2.3). Incoming events can be stored in constant time, with respect to the event-type. Events are stored sequentially in the order that they arrive, effectively sorting each list according to timestamp.

Queries execute a simple search that runs in two-dimensions — firstly through the event-types, then through the lists subject to time. Since both these axes are sorted, we can use the constraints specified by the query to cutoff the search once it leaves a valid range. However, since there is no such sorting with respect to the data value of events, queries simply check all valid events, and disregard those outside the data range. The pseudocode in Figure 5.5 demonstrates this approach.

## Iterators

It is often the case that we will only need a query to find a certain number of results. However, the number of results needed may not be known in advance — it depends upon the application, the environment and the conditions in each situation definition. We therefore have adopted an *iterator* approach for retrieving query results. Rather than waiting for a query to find all event instances, the system simply finds the first result that matches the result, and returns an iterator. Each iterator contains the result of a query, and meta-data corresponding to the original query. If more results are required, then the iterator can be passed back to find the next result.

This approach is useful for several reasons. Firstly, it saves unnecessary computation, by only finding as many results as needed. Secondly, there is no memory overhead imposed by performing a query, since no space needs to be allocated for results beyond the iterator itself. Lastly, an iterator effectively allows us to ‘pause’ a query part way through searching, which is useful for moving between queries during the detection algorithm.

## KD-Trees.

K-dimension-trees allow us to access events subject to multiple dimensions, without compromising the efficiency of any single dimension [7]. Note that in the current implementation, we have sorted events subject to type and time, but at the expense of searching subject to data. Also, while storing lists with respect to

```

EVENT-QUERY(pmin,pmax,preversed,
tmin,tmax,treversed,
dmin,dmax,dreversed)
1  if preversed
2      then pstart  $\leftarrow$  pmax
3           pend  $\leftarrow$  pmin
4      else pstart  $\leftarrow$  pmin
5           pend  $\leftarrow$  pmax
6  for i  $\leftarrow$  pstart to pend
7      do if treversed
8          then
9              node  $\leftarrow$  last
10         else
11             node  $\leftarrow$  head
12         enteredrange  $\leftarrow$  FALSE
13         while node  $\neq$  NULL
14             do if tmin  $\leq$  node.time  $\leq$  tmax
15                 then enteredrange  $\leftarrow$  TRUE
16                 if (node.time  $<$  tmin  $\vee$  node.time  $>$  tmax)  $\wedge$ 
17                     enteredrange)
18                     then BREAK
19                 if dmin  $\leq$  node.data  $\leq$  dmax
20                     then return node
21                 if treversed
22                     then
23                         node  $\leftarrow$  node.prev
24                     else
25                         node  $\leftarrow$  node.next
26 return NULL

```

Figure 5.5: The event query algorithm.

time allows us to cutoff the search once it leaves the time range, it still requires linear time to find the start of the time range.

Let us firstly consider each event to be a point in multi-dimensional space, based on the values of its properties. In SENSID we are dealing with three-dimensional points, with different axes for type, time and data. We can then consider our query to be a bounding region in 3D space, with a specified minimum and maximum value along each axis. The problem of resolving the query is then to find the event-instances (points) that are within a bounding region.

When a kd-tree is built, it performs a binary space partitioning on the set of points [7]. Each point is then stored in a single partition, up to depth  $\sqrt{(n)}$  in the tree. A point or region can be found in the tree by traversing the partitions — at each decision point, determining if the target is greater than or less than the point of partitioning on the current axis.

We can therefore use tree traversal techniques to find a reduced set of candidates that are in partitions overlapping a query region. There may still be candidates that were in the partitions, but not actually inside the region — so we must lastly filter these candidates to obtain the full result for the query

This approach enables us to find a given point in  $O(\sqrt{n} + k)$  time, where  $n$  is the number of points and  $k$  is the number of points in the result.

However, there are a few problems with this approach. Firstly, it requires additional memory, and significant time for storing and sorting events as they arrive. Secondly, we wish to retrieve the events in a particular order — this requires special consideration when building and traversing the tree. It may not always be possible to maintain search ordering across multiple axes. We have therefore chosen to use the simple query approach for now, and examine the potential of kd-trees in future work.

### 5.2.5 Detection

Given the aforementioned techniques for subscribing and storing situations and event instances, let us now look at the detection process in more detail. We will only discuss the use of the *and* operator, the most commonly used operator in situations. While our system additionally supports *sequence*, *absence*, and *counting* operators, these cannot all be covered here.

```

SIMPLE-DETECTION(S)
1  ▷ Detects if Situation S has occurred
2  tmin ← the time of the earliest lifespan of S
3  tmax ← the time detection was signaled for S
4  for o ← 0 to noperands(S)
5      do pmin ← pmax ← S.operand(o)
6          dmin ← dmax ← WILDCARD
7          if QUERY(pmin,pmax,FALSE,
                  tmin,tmax,FALSE,
                  dmin,dmax,FALSE) = FAIL
8              then return FAIL
9  SIGNAL-MATCH(S)

```

Figure 5.6: The simple situation detection algorithm

### Basic Detection

The most important factor in detecting a situation with the *all* operator is to ensure that all contributing events have occurred. Thus, our simple detection method simply goes through all event operands, and checks that there is at least one event instance for each type. Pseudocode for this simple method is shown in Figure 5.6.

This simple detection method does not take into account the consumption rules, or any conditions on the operands — although it still fulfills the primary requirement for an *all* situation to have occurred.

Before we look at how this can be extended, we will look at a few other issues. Firstly, we look at the need to queue detection requests, to maintain throughput when performing lengthy detection routines. We can store requests in a simple FIFO queue, using a singly linked version of the event storage structure. As each request is signaled, the situation to be detected is added to the end of the queue. As each detection completes, a situation is popped from the queue to begin the next detection with. We also keep a binary tag for each situation definition indicating if a detection request has already been submitted, so that only unique requests are stored.

Secondly, we need to ensure that event instances can only contribute to a single operand of a situation. We therefore use a simple tagging process in the storage component, to limit the detection of certain events. Requesting a tag

be placed on a candidate event instance will ensure that the candidate will not be detected in future queries. Tags can be retained until the event is cleared or consumed, or may be removed once the event is no longer a candidate.

Thirdly, consumption rules may be recognised by reversing queries as appropriate. When the *first* consumption rule is used, queries should search from minimum time (oldest) to maximum time (most recent) — thus finding and consuming the first, or oldest valid event. When using the *last* consumption rule, the most recent events must be examined first, so we simply tag the query to search in reverse time order. Searching using the *every* consumption rule is the same regardless of whether the query is reversed or not, although we perform consumption differently, which is discussed later.

### Advanced Detection

In order to fully support conditions in situations, we use an existing multi-purpose backtracking algorithm that has been optimised and transformed into iterative form [43]. We have translated this into a notation suitable for finding event instances subject to situation criteria and conditions. The original backtracking algorithm is shown in Figure 5.7.

To make this more applicable to our topic, we need to generalise the solution types and domains. This includes making  $t$  a variable of any type, as long as there is some way to determine an initial value, and some way to determine the next value in its domain. We also generalise the domain  $D$ , by allowing different domains for each element of the solution set. This transformed version is shown in Figure 5.8

It is now a case of translating this into a representation suitable for our application. Each problem element  $e$  is an operand definition from a single situation. The solution set  $p$  is a set of candidates that are instances of the situation operand types. We keep track of the domains  $D(e)$  by storing an iterator for each candidate. Thus, the functions *first*, and *next* are already implemented, by definition of the iterator. The function *last* simply removes and returns the last element of the solution set. When the algorithm terminates, the solution set  $p$  is either empty, or contains the participants of the detected solution.

The queries used to describe the domain(s) are in the same format as outlined in Section 5.2.4. The query event type is the type of the operand, while the query timestamp is bounded by the oldest lifespan and the time of detection. By default, there is no bounds on the data value of the events.

If there are any singular *threshold* conditions in the situation, these may place

**BACKTRACK-GP**

```
1  ▷ A multi-purpose backtracking algorithm
2   $p \leftarrow \emptyset$ 
3   $t \leftarrow 1$ 
4  while  $p \neq \emptyset \vee t \leq D$ 
5      do
6          if  $t > D$ 
7              then  $t \leftarrow \text{LAST}(p)$ 
8                   $t \leftarrow t + 1$ 
9          elseif  $V'(t,p)$ 
10             then if  $C'(t,p)$ 
11                 then  $\text{PROCESS}'(t,p)$ 
12                      $t \leftarrow t + 1$ 
13                 else  $p \leftarrow p \cup \{t\}$ 
14                      $t \leftarrow 1$ 
15             else
16                  $t \leftarrow t + 1$ 
```

Figure 5.7: A multi-purpose backtracking algorithm, where  $p$  is a partial solution set,  $V(p)$  is true when  $p$  is a valid solution,  $C(p)$  is true when  $p$  is a complete solution set,  $process(p)$  is a routine to handle the complete and valid solution  $p$ ,  $t$  is an instance of an element of the solution set, and  $D$  is the domain of  $t$ . Note that  $V'(t,p)$  is the same as the expression  $V(p \cup \{t\})$ , similarly for  $C'(t,p)$  and  $process'(t,p)$ . the function  $last(p)$  removes and returns the last element of the set  $p$ . Also, in this multi-purpose version,  $t$  is represented as integers, and  $D$  is an upper limit.

BACKTRACK-TRANSFORMED

```

1  ▷ A generalised backtracking algorithm suited for our purposes
2   $p \leftarrow \emptyset$ 
3   $e \leftarrow 0$ 
4   $t \leftarrow \text{FIRST}(D(e))$ 
5  while  $p \neq \emptyset \vee t \neq \text{NULL}$ 
6      do
7          if  $t = \text{NULL}$ 
8              then  $t \leftarrow \text{LAST}(p)$ 
9                   $e \leftarrow e - 1$ 
10                  $t \leftarrow \text{NEXT}(t, D(e))$ 
11         elseif  $V'(t,p)$ 
12             then if  $C'(t,p)$ 
13                 then  $\text{PROCESS}'(t,p)$ 
14                      $t \leftarrow \text{NEXT}(t, D(e))$ 
15                 else  $p \leftarrow p \cup \{t\}$ 
16                      $e \leftarrow e + 1$ 
17                      $t \leftarrow \text{FIRST}(D(e))$ 
18             else
19                  $t \leftarrow \text{NEXT}(t, D(e))$ 

```

Figure 5.8: The transformed backtracking algorithm, where  $e$  keeps track of the current element that is being resolved, and  $D(e)$  is the domain of element  $e$ . Additionally, the function  $\text{first}(D(e))$  selects the first element of a domain, and  $\text{next}(t, D(e))$  selects the next element of a domain, given the current element  $t$ . If there are no more elements in the domain,  $\text{next}(t, D(e))$  returns NULL.

additional bounds on the query, possibly adding a lower and/or upper bound on the data field, or further constraining the bounds of the timestamp.

The function  $C'(t, p)$  simply checks if the solution set  $p$  will be complete when  $t$  is added — this can be done by checking if  $e$  (or the size of  $p$ ) is one less than the total number of operands in the situation . The function  $V'(t, p)$  checks if the solution set  $p$  will continue to be valid when  $t$  is added. This is done by performing a partial evaluation of  $p$ :

**Partial Evaluation:** Find the set of conjuncts  $C$  in the situation that involves the current operand  $e$ . For each conjunct  $c$  in  $C$ , evaluate the expression using the binding  $t$ , plus the bindings specified in the solution set  $p$ . If there are operands in  $c$  that do not yet have a binding in  $p$ , assume the expression evaluates to *true*. If all conjuncts in  $C$  are *true*, then the partial evaluation is *true*, otherwise it is *false*.

### 5.2.6 Post-Detection

Once a situation is detected, there is a series of post-detection steps that must occur.

Firstly, an *internal event* must fire immediately, signaling to the dispatcher that a situation has occurred. The speed of delivery of this event is important, as there may be nested situations that depend upon the internal event. Secondly, the client application is notified of the situation occurrence by a separate event fire. The client may use pointers from the event-fire to copy situation participants to application memory for later processing. Finally, once the client has returned from the event handler, event consumption occurs, freeing the memory used by them. For the *first* and *last* termination quantifiers, it is simply the participating events found during detection that are consumed. For the termination quantifier *every*, the backtracking algorithm runs until all valid solutions are found, and consumes each one.

## CHAPTER 6

# Results: Statistics and Measurements

This chapter looks at the tests we have performed with SENSID, and discusses the systems' performance, memory usage, and expressiveness.

## 6.1 Tests

### 6.1.1 Implementation Environment

In order to test the system, we built sample applications for both the TinyOS simulator and mica2 motes.

The SENSID system itself is written in pure NesC. The only TinyOS dependency is timestamping, and can be changed to suit application needs. Thus, this is effectively multi-platform *middleware*, as long as the required platform specific event producers are supplied.

### 6.1.2 Event Producers

Since it is difficult to repeatably produce series of complex, real-world events, we used a set of proxy event producers for testing. These event producers use the same interface expected by real event-producers in mote deployments, intermittently firing events with associated data. To perform a basic simulation of events that may occur in the real world, we use random timers or scripted sequences that determine the order that events are produced.

### 6.1.3 Simple Application

A sample, statically programmed application was built to test the basic principles of the system, as well as for checking performance. This application contains all

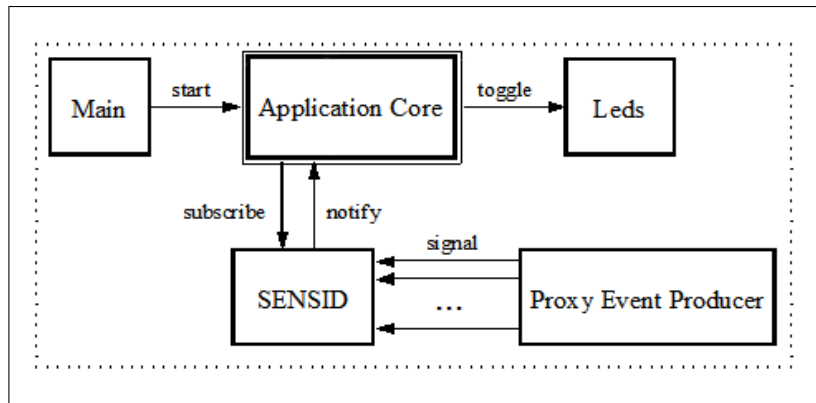


Figure 6.1: The design of a simple application, utilising SENSID as a code utility. On booting the mote, the *Main* component loads the application, which subscribes to static situations through SENSID. When it receives notifications of a situation occurrence, it toggles the on-board LEDs.

situation definitions built into the code, and on booting the mote, it immediately subscribes to them with SENSID. When the application receives notification of a matched situation, it notifies the user through simulator print statements or mote LEDs.

While simple and straightforward, this application demonstrates the use of SENSID as a form of code-utility. Rather than integrating with remote systems and running dynamic scripts, it simply serves its purpose of reducing complex composite event code into simple situation definitions. Figure 6.1 illustrates the configuration of the sample application with SENSID.

#### 6.1.4 Maté

To better examine the potential of SENSID, we have developed a set of suitable hooks into the Maté Virtual Machine environment. This allows us to build a virtual machine that uses SENSID as a event-handling back end, and Maté contexts and opcodes as an execution environment. A diagram of how SENSID fits in with the Maté framework is shown in Figure 6.2.

*Opcodes* and *contexts* are the two main execution primitives in Maté [34]. Opcodes form the basis of a programming language, and encapsulate high level application specific operations. Contexts define an execution context, such as event handlers. A typical use of Maté is as follows:

A VM descriptor file is written, including all contexts and opcodes that are

to be available in the virtual machine. A script then brings together all the required TinyOS code, and compiles the VM into a single binary image that can be deployed or run in a simulator. Users can then define the system's responses to events using a language compiler such as TinyScript. This lets the user pick a target context, (such as *Once*, *Timer*, *MessageReceived*), and enter BASIC-like instructions. The code is then parsed, compiled, and propagated through the network. For more details, refer to the full Maté and TinyScript manuals [32, 33]

In our approach, we remove the need for individual contexts that are only capable of handling single events. Our SENSID VM descriptor file includes all of the required opcodes for subscribing to situations, as well as a set of generic *Action* contexts. There is also a set of opcodes corresponding to events types that SENSID can handle. Using these opcodes in a VM descriptor file implicitly binds event-sources to the SENSID middleware, so they can be used in situation subscriptions.

At run-time, users can use TinyScript to load code into a *Subscribe* context. This runs the required SENSID functions to subscribe to a new situation. When a situation match occurs, the *Action* context associated with the situation is triggered. Users can implement responses to situations by loading application specific handling code into the correct action context.

## 6.2 Resource and Performance Statistics

### 6.2.1 Code Size

The prototype SENSID implementation has an efficient code representation, amounting to some 1600 lines of NesC code. The simple test application was able to be built in as little as 90 lines, while the Maté hooks require an additional 350 lines, plus configuration files (Table 6.1). By desktop software standards, this is a modest size for a code utility. However, due to issues of software complexity, and the relative infancy of WSN programming, most existing TinyOS libraries have a small code base. The current single largest component is the TinyDB application, consisting of 4000 lines of code [23].

### 6.2.2 Code Memory Footprint

The breakdown of the compiled code size is outlined in Table 6.2, showing the ROM usage for each component. The first ROM column shows unoptimised code

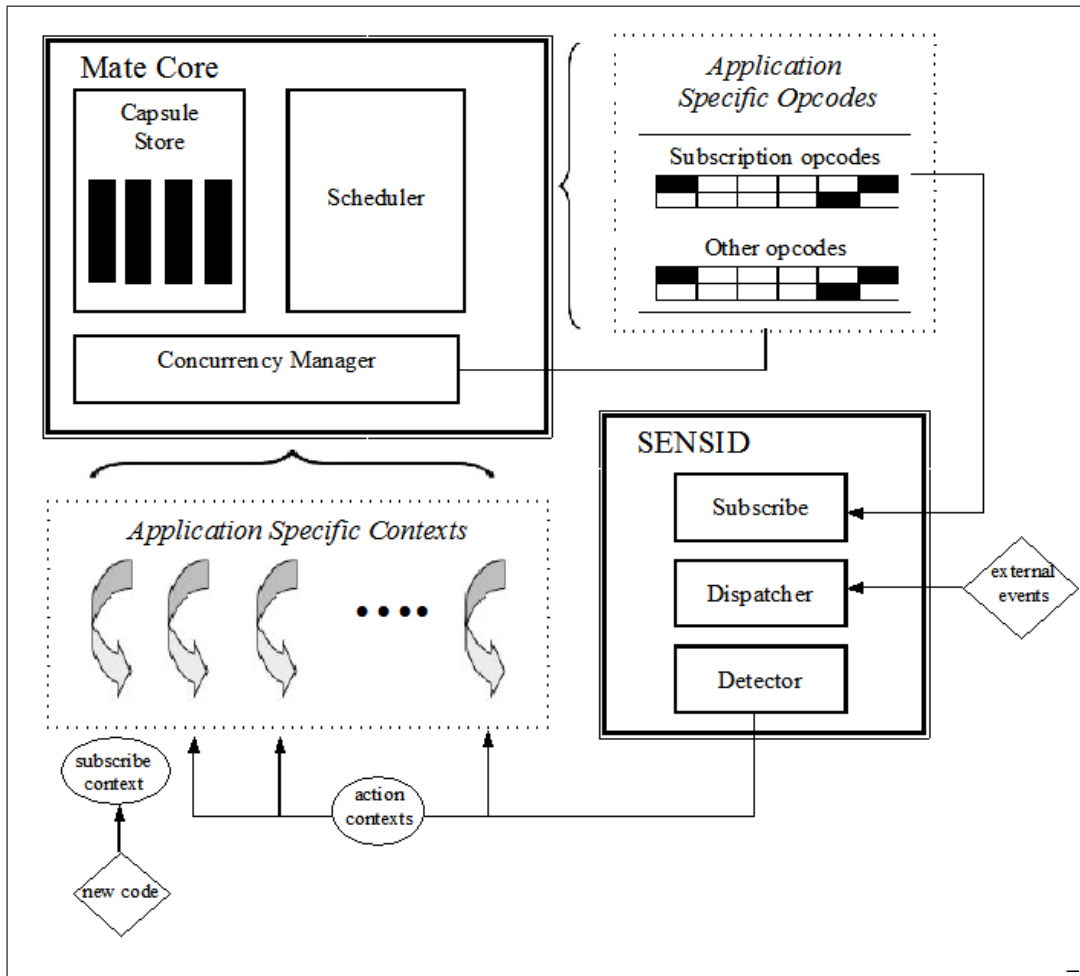


Figure 6.2: Integration of SENSID into the Maté virtual machine framework. SENSID provides a set of opcodes to support subscription to situations from within Maté. *Action* contexts are provided to execute Maté code in response to a situation occurrence.

Component	Lines of NesC
Core	1600
Simple Application	90
Maté (hooks only)	350

Table 6.1: Size of SENSID NesC code

Component	ROM (bytes)	ROM, -Os (bytes)	RAM (bytes)
Subscribe	246	39	4
Dispatcher	962	1484	10
LifespanManager	1708	266	7
OperandFilter	685	106	1
OperandStorage	2176	509	18
Detector	3040	1354	6
<b>SENSID</b>	<b>8817</b>	<b>3758</b>	<b>46</b>
Sample Event Producers	750	108	2
TinyOS Components	7106	2595	69
Maté — Core+TinyScript	~90000	24496	1213
Maté — SENSID	~2000	987	10
<b>Simple Application</b>	<b>16673</b>	<b>6461</b>	<b>117</b>
<b>Maté Application</b>	<b>~108000</b>	<b>31944</b>	<b>1340</b>

Table 6.2: Compile-time memory costs per component, compiled with both no optimizations (-O0) and size optimisation (-Os).

sizes, while the second shows the size after using the NesC compilers *optimise for size* (-Os) option.

Unsurprisingly, in the unoptimised code, the most significant component is the situation detector, followed by the operand store (with query algorithms), and lifespan manager (with garbage collector). Notice that the total code size of SENSID is not much larger than the TinyOS core components, which includes only the scheduler, timers and timestamper, and power management. Overall, the entire system, capable of representing such situations as demonstrated in Section 6.3, fits unoptimised in a small package of less than 17 kilobytes.

Comparing this against the size optimised code, we see a substantial reduction in the code footprint. Due to the code inlining performed by the NesC compiler, it is difficult to see the per-component breakdown — for example the majority of operand filtering and lifespan operations are inlined into the dispatcher functions. However, we can see that the total SENSID core now has a footprint of less than 4 kilobytes, and a complete application size of less than 7 kilobytes. Considering that current generation mote technology has approximately 64–128 kilobytes of program memory, and even the smallest smart-dust technology includes 32 kilobytes of ROM, this footprint is quite acceptable.

Type	Data (bytes)	Structural (bytes)	Total (bytes)	Optimized (bytes)
Condition	8	2	10	7
SDN	3	6	9	8
Event Instance	6	4	10	10
Lifespan	4	6	10	10
Situation	7	6	13	11
Maté Event	-	-	-	7
Maté Context	-	-	-	192

Table 6.3: Allocated Memory Costs (by type, per unit, in bytes). Optimizations include simple-tricks like bit-shifting groups of variables together.

### 6.2.3 Runtime Memory Usage

While fitting the SENSID code into the confines of mote program memory has been relatively straightforward, maintaining the system state within the limitations of 4 kilobytes of RAM is more challenging. Looking at Table 6.2, we see that the memory overhead for SENSID has been kept to a minimum, consuming only 46 bytes of memory. However, the major consumer of RAM is the space used in storing dynamic data structures — essentially the situations, lifespans and events. The unit costs of storing each of these dynamic types in shown in Table 6.3.

To examine the total RAM costs of running the SENSID system, we will consider a typical build. For example, a system requiring 5 situations (65B), 10 conditions (100B), and 25 bindings (225B) will require 390 bytes for subscription purposes. This allows each situation an average of 1 initialisor, 1 terminator, and 3 operands — although any other distribution is possible. For run-time storage, we may decide to allocate enough storage for 25 lifespans and 75 events, requiring 1000 bytes of memory. Thus, a typical system such as this can be built requiring a total of only 1390 bytes for the SENSID middleware.

Coupled with Maté, using the overhead from Table 6.2, plus an additional 192 bytes per action context, a complete Maté build could be made for 3690 bytes of memory. This leaves enough remaining room for sensor-drivers and event producers, to fit within the 4 kilobyte limit.

There are already new mote platforms such as the Telos mote (rev. B), offering additional memory beyond the current limitations [42]. As mote memory sizes improve, the sizes available for the dynamic memory store will grow, allowing much more complex situations and larger solution domains.

## 6.2.4 Speed Performance

It is difficult to perform accurate, repeatable tests using the mote hardware itself, especially when it comes to triggering precise patterns of events and delivering reliable feedback. Thus we used two different computer simulation approaches to run SENSID on a normal desktop PC.

**TOSSIM** is a TinyOS SIMulator, and uses the NesC compiler to compile code into a PC binary [35]. The key advantage of this approach is that it can run code using machine instructions native to the PC platform, allowing simulations to run at very high speeds. However, due to differences between implementations of native instructions and mote instructions, measurements such as machine time and clock cycles are unreliable. It is useful, however, for studying the behaviour of patterns of events, or using performance heuristics such as loop counting.

**Avrora** is an AVR-processor interpreter, capable of interpreting instructions from mote-compiled code at the instruction level [51]. Using disassembled binaries of the sample applications, this is able to perform a perfectly accurate simulation of the program, and use features such as breakpoints and stack-checkers to perform analyses. Avrora is extremely useful for counting the clock-cycles required for key functions, and given the mote processor clock speed, translate it into expected execution times.

### Event Throughput

We firstly looked at the event throughput of the system, by measuring the number of clock cycles spent in component functions using Avrora. Table 6.4 shows a summary of some of these results. Unsurprisingly, the time required for these operations is negligible, with most operations taking less than one tenth of a millisecond to run on a mote's 7MHz processor. SENSID could therefore sustain a high rate of event throughput (disregarding limitations of memory and detection speed). Even if an event is dispatched to each component multiple times, event throughputs of up to 1000 events per second are possible — far greater than the requirements for any existing network.

### Detection Algorithm

We have used two ways of measuring the performance of the detection algorithm — clock cycles, and the number of solution nodes (combinations of candidates) visited during the backtracking. Number of nodes visited is a useful metric, since we can simply use code counters, and run simulations at full-speed in TOSSIM.

Component	clock cycles
Event Dispatch	223
Lifespan Open	319
Lifespan Termination	628
Operand Filter	370
Operand Storage	257
Detection Queuing	218

Table 6.4: Performance of events in system, broken down by component. Event dispatching is the overhead only, lifespan termination does not use garbage collection, and operand filtering only tests against a single condition.

Each of our tests uses a single situation, with one initialiser, one terminator,  $n$  operands, with the *and* operator and *deferred* detection mode. The search space is set up by a single *init* event, followed by a sequence of events that is evenly distributed between each event type. The size of the search space is defined by the parameter *epo*, which determines the number of *events per operand*. Finally, a lifespan terminator event is fired, triggering the detection algorithm. Thus a simple test, with parameters  $n=3$ ,  $epo=2$  would use the event sequence  $\{init, 1, 2, 3, 1, 2, 3, term\}$ .

Our test scenarios tests several different size search spaces, from small numbers of events ( $epo=4$ ), to large numbers ( $epo=20$ ). For each search space, we cover a range of possible situation definitions, from  $n=1$  to  $n=8$  inclusive, noting that it is unlikely that sensor network applications will require more than eight operands in a single situation definition. Each test measures both the number of cycles and the number of nodes visited for the *worst case* execution of the algorithm. Worst case execution is defined as covering all possible combinations through the backtracing procedure — that is, every event-instance is examined, and yet no combination meets the criteria for the situation to be complete.

Figure 6.3 shows the results for some of these tests, with search spaces of  $epo= 4,6,10$  and  $20$ .

Using an exponential scale, we can see immediately that the algorithm is NP complete, as noted in existing literature [15]. Note that the number of nodes visited during detection reflects a depth-first tree traversal algorithm, of depth  $n$  and with branching factor *epo*. Thus, the time taken to reach each node reflects the total number of nodes in the tree. This is found by calculating the total number of nodes in the tree:

$$numnodes = epo^n + epo^{n-1} + \dots + epo + 1$$

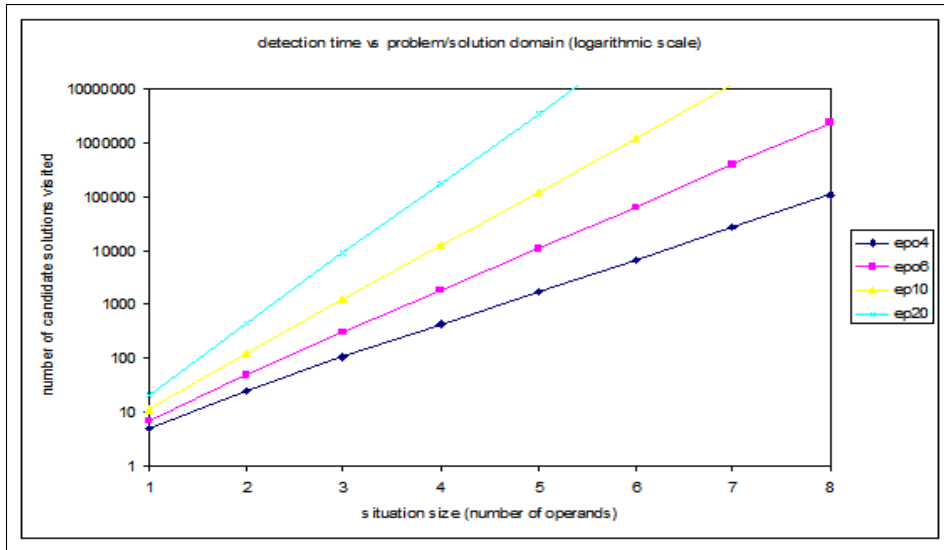


Figure 6.3: Number of nodes visited during detection, with respect to situation definition size. Note the exponential scale.

However, we also include the time taken to backtrack each event — therefore, each node must be counted twice, except for the last row:

$$numnodes = epo^n + 2 * (epo^{n-1} + \dots + epo) + 1$$

Thus, the worst case time complexity is  $O(epo^n)$ , with execution times exponential with respect to the size of the situation definition. Table 6.5 shows how this translates to the number of clock cycles required to perform detection, and thus the time required if the code were running on a mote. We see that the times are acceptable for real-time response when the number of operands is less than 6, although the performance rapidly deteriorates if the problem size grows any larger.

However, this is not a major concern for wireless sensor networks, since the few applications that require a large number of operands (such as environmental monitoring) do not usually have real-time requirements. This gives the system an acceptable time restriction to work with, where detection times of a few minutes is still feasible.

Now, assuming that the size of a situation is fixed, the worst case time complexity is  $O(epo^c)$ , for some constant number of operands  $c$ . This gives a polynomial number of solution nodes to visit, subject to the size of the solution domain. To see how this affects the actual execution time, we use fixed situation size ( $n=4$ ), and see how worst case detection times vary over different values of

operands	detection nodes	clock cycles	time (ms)
1	5	1479	0.2
2	25	9170	1.2
3	105	35723	4.8
4	425	142098	19.2
5	1705	567135	76.9
6	6825	2267676	307.6
7	27305	9069641	1230.6
8	109225	36277450	4922.3

Table 6.5: How the number of nodes reflects on actual time used, with a fixed problem size of 4 events per operand. Times are calculated using a Mica2 clock speed of 7.37 MHz.

events per operand	detection nodes	clock cycles	time (ms)
4	425	142098	19
8	5265	1542386	209
12	24505	6793050	921
16	74273	19974130	2710
20	176841	46654882	6330
24	360625	93892890	12739

Table 6.6: The effect of solution size for a fixed situation size of 4 operands. Times are calculated using a Mica2 clock speed of 7.37 MHz

*epo.* The results of this test is shown in Table 6.6, along with the actual time requirements.

We lastly decided to investigate how the number of conditions affects the running time of the detection algorithm. Surprisingly, the number of conditions doesn't *directly* affect the performance very much. Table 6.7 shows that each additional condition simply adds a constant overhead to the cost of examining each node, which constitutes a linear increase on top of the existing costs. However, additional conditions may impose tighter constraints on each event instance, thus indirectly increasing the failure rate of each candidate. As failure rates increase, the number of nodes examined *may* increase, and thus worsen performance. This does not show up in this test, as we are already assuming *worst case* performance, where the number of nodes examined is always at a maximum.

operands	detection nodes	clock cycles (1c)	clock cycles (2c)	clock cycles (4c)
1	5	1479	1579	1779
2	25	9170	9766	10766
3	105	35723	40543	45127
4	425	142098	156134	176334
5	1705	567135	618183	669103
6	6825	2267676	2466328	2812604
7	27305	9069641	9858857	11386261
8	109225	36277450	39429070	45681134

Table 6.7: The effect of different numbers of conditions on actual CPU-time, with fixed solution domain of 4 events per operand.

### Hybrid Approach

We have stated that most WSN applications have a small number of operands and event instances. However, there may be some unforeseen application that use complex situations over large domains. Given the enormous computation times in these circumstances, we look at a number of alternative approaches.

The first, more preferable alternative is to find a way that situations may be subdivided. If there are operands in a situation that are not interrelated (i.e. there is no set of conditions connecting them), some may be moved to a new situation with the same lifespan class. The branched situation can use an internal event from the first situation to ensure that *both* sub-divided situations are satisfied. Assuming an even subdivision, this reduces the single search tree of depth  $n$  to two search trees, of approximate depth  $n/2$ . Thus, the time complexity is reduced to  $O(eps^{n/2})$ .

This may not be suitable for all applications, especially where there is a high degree of interrelation between operands, or when situation sub-division does not significantly reduce detection times. In this case, it may be necessary to offload some of the computation onto a central store, in a similar manner to sensor network databases like TinyDB. Sensor network nodes can still perform operand filtering, lifespan control, and detect small sub-situations, but the results are forwarded to a database for the detection of the complex situation(s). Unfortunately, this prevents the network from reacting to situations as they occur — but we retain the benefit of reduced data transmissions over pure database approaches.

A complete analysis of these hybrid approaches is left as a subject for future work.

## 6.3 Expressiveness

In this section, we compare the expressiveness of the prototype against other systems. We firstly look at how SENSID compares with AMIT, to see how the expressiveness is affected by the translation to sensor networks. We also compare SENSID against existing composite-event models for WSNs. We do not look at how it compares against active database approaches such as ODE and Snoop, as this is better covered by various AMIT reports [5, 6].

### 6.3.1 Comparison vs AMIT

There are essentially three main restrictions in the SENSID implementation versus the original AMIT implementation:

- No support for named data attributes and conditions
- No event groupings or data keys
- A simplified storage model

#### Named Event Data

AMIT supports the association of named data attributes with each event type, and supports conditions over these schema. SENSID does not support these features, although it does not have a significant impact on the sort of WSN applications that can be specified. Most event producers are sensors, which usually have only a single type associated with them. In particular, most sensors supported by TinyOS use a standard interface, which uses typed 16-bit integer data. We have adopted this format for events, and thus SENSID can use most TinyOS sensors as event producers.

There are still some forms of events that contain multiple pieces of information, for example the receiving of a network packet. Generally, SENSID should not be used for complex networking models, since more efficient implementations can be provided at a lower level. However, when features such as networking are required for a situation, it may be possible to abstract the essential meaning of the packet into an event type, and use the single event data to convey the most significant piece of information. For example, a situation that needs to know when certain nodes have reported some situation, can use event abstractions such as *receive\_situation\_x(node)*, where *x* is the situation reported, and *node* is the source of the packet.

## Event Groupings

The other major restriction over the AMIT event model is the lack of event groupings and data-keys. AMIT supports the grouping of events into equivalent types, so that a situation may support a set of events as a single operand. Data keys are used to denote attributes that are equivalent between events of the same group, to allow condition to be specified over an event group.

Data keys are not required in SENSID, since it does not support multiple event attributes. Also, event groupings are not usually required in sensor networks, due to a clear distinction between the different event producers.

However, if event groupings are required, they may be supported at the event-producer level. Using the NesC component model, extra event-ports can be added to the SENSID dispatcher — these are the ports representing an *event group*. By connecting the interfaces of several event types to these ports, the event types effectively become members of the same group. The event group can then be used in any situation definition as if it was a normal event-type, while the events are actually received from multiple sources.

## Simplified Storage Model

The move to simplify the storage model for SENSID was largely a matter of memory restriction, since even the smallest embedded databases are better suited to devices like palm-top computers. AMIT uses features such as a full relational DBMS, and selection and joining operators to provide different views of events — according to types, groupings, situations and lifespans. These features help the implementation and performance of the various situation detection operators. While we have greatly simplified how events are stored and accessed in SENSID, it has not impacted on the expressiveness of the system.

### 6.3.2 Comparison vs Composite Events in WSNs

We now compare SENSID against existing WSN composite-event detectors. Generally, interval-based approaches such as DSWare are good at specifying systems that need a variety of simple lifespans. SENSID is more suited to complex lifespan definitions, nested situations, and situations involving absence and counting operations. We will therefore look at a few different examples, and will look at how they may be expressed with both approaches.

## Explosion Detection

Explosion Detection is an example real world event used by several approaches in testing specifications and implementations [37, 45]. It looks at how a sensor can detect the occurrence of explosions in an external environment, by using sensors to detect light, sound and temperature.

### Problem

Find when extremes of light, sound, and temperature occur within a set period of time. Specifications may use sensor abstractions such as “*bright*”, “*loud*”, and “*hot*” that indicate when readings are above a threshold.

### *DSWare*

A specification for this problem would be something like:

$$e = (\textit{bright}, b) \wedge (\textit{loud}, l) \wedge (\textit{hot}, h)$$

where  $b, l$ , and  $h$  are the detection intervals after the occurrences of *bright*, *loud*, and *hot* respectively.

Under some circumstances, we may choose to use the same time interval for each event, so that when any event occurs, the remaining events must trigger within ( $b = l = h$ ) seconds. A more realistic definition of the problem may require a more scientific ordering of the events — such that light must occur before sound, which occurs before heat. While we cannot strictly enforce this requirement, we may approximate it by using different intervals, such that  $b > l > h$ .

### *SENSID*

If we use the assumption that  $b = l = h$ , we are able to capture the basic requirements with a single situation definition as shown in Figure 6.4. If we were to capture the sequential nature of the events in an explosion, we can use the same approach, but instead use the *sequence* operator. Using operators to define and change behaviour life this is more direct and easier to understand than adjusting multiple time intervals in *DSWare*.

If we need to specify unique validity intervals for each event (like *DSWare*), we can define additional conditions on the timestamps of each operand. We can also capture more detailed requirements, such as “*loud* must occur less than two seconds after *bright*, but at least 3 seconds before *hot*”. Figure 6.5 shows how this can be represented in a situation definition

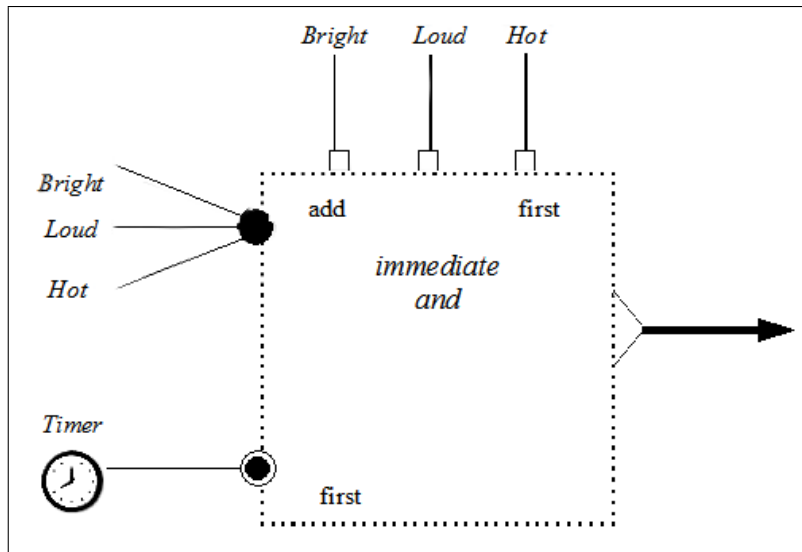


Figure 6.4: Situation diagram for explosion detection

Plant at risk

The *plant-risk* example was covered in Section 3.4.3 as a demonstration of situation specifications. We will compare our situation definition approach (Figure 3.2) with a possible solution using DSWare.

*DSWare*

**Finding soil-moisture drop:** DSWare has no facilities for defining sequences of events with conditions between them. Therefore, finding a drop in the soil-moisture must be implemented as a primitive event type. However, with this approach, the threshold that determines the significance of a drop must be embedded into the event producer. It is preferable to use SENSID's approach where thresholds are changeable according to the situation definition.

**Finding dry/no rain:** This is impossible to specify properly in DSWare. We can specify an interval such as

$$dry = (sm, 1 \text{ week}) \wedge sm < dry\_threshold$$

however we cannot specify the *absence* of rain during this period. We would instead have to capture this sub-situation as a primitive event. Such an event would start a timer on *dry* readings, cancel the timer if rain occurs,

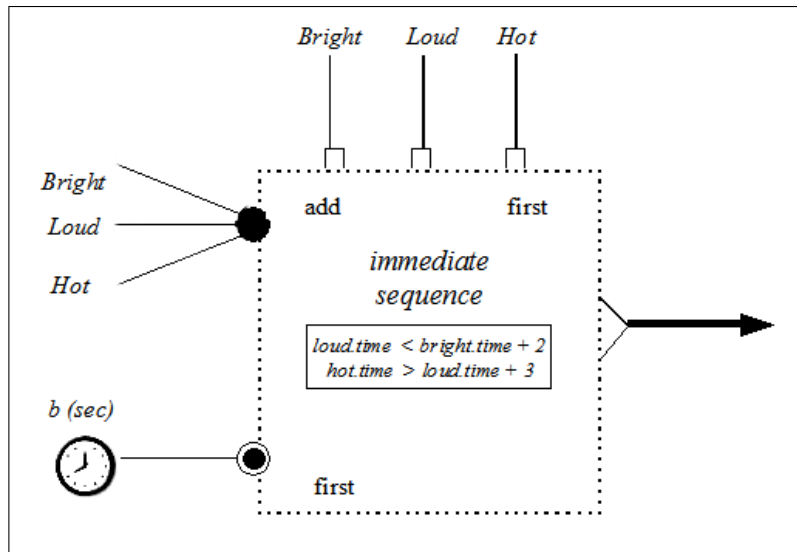


Figure 6.5: Situation diagram for explosion detection, with additional temporal constraints

and fire a *no-rain* event when the timer expires. Again, this is undesirable, because the specification is embedded in the event-producer and cannot be easily changed once the system is deployed.

**Finding plant-risk:** This occurs when both a *no-rain* event and *soil-moisture* event occurs together. We may use a specification such as

$$\text{plantrisk} = (\text{norain}, 1 \text{ day}) \wedge (\text{smdrop}, 0)$$

however this does not fully capture the requirements, as it uses a fixed interval of one day. Specifically, we are looking for *sm-drop* events that occur after a *no-rain* event and before *rain* event (if one exists). This is impossible to specify in DSWare, since it cannot use events to define the end of a temporal context.

### Traffic Detection

The problem of detecting traffic situations is a challenging application for sensor networks. There are a multitude of different ways of specifying temporal interactions between sensors. We will use a simple abstraction of the problem, although it remains sufficiently complex that it is difficult to specify using an interval-based approach such as DSWare. We will focus on how SENSID is able to capture the requirements of the problem.

Problem

Find occurrences of traffic accidents that lead to traffic jams, and reroute traffic. If the traffic does not clear, dispatch a traffic team.

*SENSID*

We have created one possible specification for this problem, shown in Figure 6.6. There are a number of key features in this example, many of which cannot be reproduced using interval methods such as DSWare.

**Detection of traffic levels:** We have used the SENSID counting operators to obtain events based on the (conditional) number of occurrences of events. Thus, finding when there is a traffic jam may be specified by finding when the number of cars passing the sensor (per minute) drops below a certain point, and all are travelling at a slow speed. Finding when the traffic clears can be specified in a similar manner.

**Cause and effect:** Identifying the *accident* event, and *traffic jam* internal event as an ordered sequence (within a time-limit) allows us to infer when the accident has lead to a traffic jam.

**Actions:** When the traffic-from-accident is first detected, SENSID notifies the application. Using (for example) Maté, we could notify another system that allows traffic to be redirected.

**No effect?:** Using the absence operator *unless*, we are able to find situations where rerouting has occurred, but the traffic jam has not cleared.

The Mate code for this program specification, and subsequent actions are available in Appendix D.

Summary

We have explored a number of example real-world composite events, and how they can be defined using both DSWare and SENSID. We have shown that DSWare is only able to specify the examples to a limited extent. While capable of defining simple expressions using time boundaries, applications involving absence of events, counting of events, or specific temporal ordering are beyond its scope.

We have shown that SENSID is not only able to represent the definitions possible in DSWare, but also offers a greater degree of expressiveness for capturing more complex requirements.

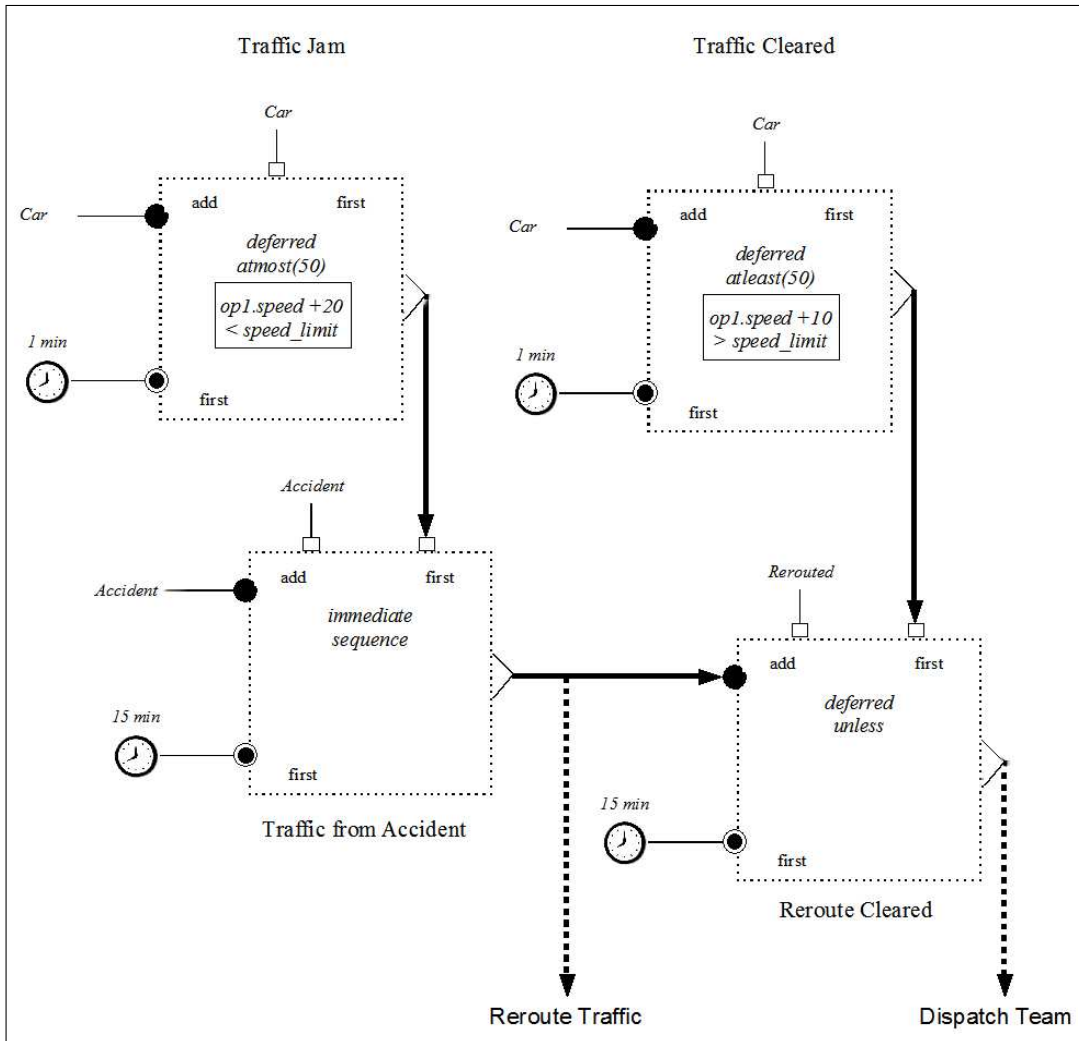


Figure 6.6: Situation diagram for traffic detection

## CHAPTER 7

# Discussion and Conclusions

Sensor network applications that are required to react to real world events need to detect event patterns within the network. This reduces the bandwidth and energy costs of communication, by only transmitting high level notifications, rather than large quantities of raw data.

DSWare has previously explored this problem, addressing the correlation among different sensor observations and the inherent, real-time characteristics of sensors [37]. Similarly, Römer and Mattern have stressed the need for sensor networks to be capable of detecting certain real-world states [45]. However, current ideas do not allow the expression of complex and conditional relations between event types, and it can be difficult to capture the temporal and spatial characteristics of event-patterns that applications need to know about.

We have presented a novel approach to the problem of event-detection, using the idea of subscribing to *situations* to capture important details event patterns. Based on concepts explored in active-databases, a situation definition features a richer language than current sensor network approaches — incorporating concepts such as contexts of relevance (lifespans), relation operators (such as joining, counting, absence and temporal), event data conditions and situation nesting. We believe that these features make the situation notation uniquely suited to specifying complex real world scenarios, traffic conditions.

The core of our work has involved the investigation of how well the semantics of situation detection translate to sensor network platforms. Our prototype SENSID is extremely promising — it has shown to be capable of handling a wide range of situations that were not possible with other WSN approaches. One of our main contributions is determining the benefits of making tradeoffs between the expressiveness of the specification and the efficiency of representation. We have claimed that many features of the original AMIT system, such as support for conditions over named event attributes, are not required for the majority of sensor network applications. This is because in WSNs, events are typically from sensors and record data of a single type. Thus, features such as these may be

rejected in favour of a more succinct representation. This, in addition to the careful selection of memory-conserving data structures, has enabled us to substantially reduce the footprint and run-time memory requirements of SENSID in comparison to AMIT. Our prototype is capable of storing complex situation definitions in fewer than 100 bytes of memory, and records state information such as event instances and lifespans in only 10 bytes each. Coupled with the ability to tune storage ratios subject to the application domain, the system is able to handle moderate size problem domains within the tight confines of a mote's 4 kilobytes of memory.

The other key contribution of our work has been revealing how well situation detection middleware performs under the limited processing abilities of mica2 mote hardware. We found that the processes of dispatching events and opening lifespans were not of any concern, given the optimised code and data structures. Storage and retrieval of events performed well, due to its simplicity and small domains, although a better solution for future implementations using KD trees has been proposed. We discovered, however, that the performance of the detection algorithms remains the most challenging issue. While simply finding events to match the situation definition is relatively straightforward, resolving (potentially conflicting) conditions on each one is a complex problem — related to fields such as the notorious Boolean satisfiability problem [17]. We chose to use a backtracking algorithm as in the AMIT system, since it is both a correct and complete solution, and were able to adapt an efficient, general purpose approach to suit our needs. The algorithm remains NP-complete, such that worst-case execution time scales exponentially with respect to problem size and domain size. However, most applications have small problem sizes, with 5 or fewer event types contributing to a single situation. Also, since situations are able to manage their lifespans, we can effectively control the size of the solution domain. Our results indicate that reduction of either of these variables gives an acceptable execution time for many applications — particularly fields such as environmental monitoring where short delays (seconds to minutes) are acceptable. We have also discussed the possibilities of sub-dividing situations for better performance, or delegating some situation detection to an offline, out-of-network operation, although exploration of these tradeoffs is beyond the scope of our current work.

There are many interesting directions that future work in this area can take. There is still much work to be done with respect to efficiency, such as improving the garbage collection algorithms, and improving event retrieval queries (possibly using the suggested KD-tree approach). There are also possibilities such as using mote logging abilities to offload event instances during long-running lifespans. In terms of the detection algorithm, there are many improvements yet to be investigated, such as refining event queries subject to the conditions in order to

reduce the size of the search space. Other topics include studying the effect of ordering on backtracking, and determining if it is possible to perform partial detection to incrementally expand a backtracking algorithm as events arrive. Lastly, there are further refinements that could be made to the specification, namely the inclusion of spatial operators [4]. It is unclear how spatial operators can function over a distributed (and unreliable) network, and therefore work in this area will need to consider many underlying factors of wireless sensor networks.

The SENSID system demonstrates a feasible and practical approach for detecting complex situations in sensor networks. We are only beginning to see the true power of wireless sensor networks and, by example alone, we see that specification of complex event patterns will play an important role in realising the possibilities. Our prototype has shown that moderately complex situations with temporal interrelations can be expressed simply, detected efficiently and acted upon — all within a sensor network environment.

## References

- [1] The family of motes preceding Telos and their capabilities. Reference Material, Retrieved from <http://webs.cs.berkeley.edu/papers/hotchips-2004-mote-table.pdf>, Aug. 2004.
- [2] avr-libc Reference Manual 1.2.3. User Reference Manual, Retrieved from <http://hubbard.engr.scu.edu/embedded/avr/doc/avr-libc/avr-libc-user-manual/>, Feb. 2005.
- [3] ADI, A., BIGER, A., BOTZER, D., ETZION, O., AND SOMMER, Z. Parallel implementation of composite events. In *22nd International Conference on Distributed Computing Systems Workshops* (Vienna, Austria, July 2002), ICDCSW 2002, pp. 579–581.
- [4] ADI, A., BIGER, A., BOTZER, D., ETZION, O., AND SOMMER, Z. Context Awareness in Amit. In *Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services* (Seattle, Washington, USA, June 2003), AMS 2003, pp. 160–166.
- [5] ADI, A., AND ETZION, O. The Situation Manager Rule Language. In *The First International Workshop on Rule Markup Languages for Business Rules on the Semantic Web* (Chia, Sardinia, Italy, June 2002), RuleML 2002, pp. 49–64.
- [6] ADI, A., AND ETZION, O. Amit — the situation manager. *The VLDB Journal* 13, 2 (2004), 177–203.
- [7] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.
- [8] BERGBREITER, S., AND PISTER, K. CotsBots: An Off-the-Shelf Platform for Distributed Robotics. In *Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems* (Las Vegas, Nevada, USA, Oct. 2003), IROS 2003, pp. 16–32.
- [9] BONNET, P., GEHRKE, J., AND SESHADRI, P. Querying the Physical World. *IEEE Personal Communications* 7 (Oct 2000), 10–15.

- [10] BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. B. Design and Implementation of a Framework for Programmable and Efficient Sensor Networks. In *The First International Conference on Mobile Systems, Applications, and Services* (San Francisco, CA, 2003), MobiSys 2003, pp. 187–200.
- [11] CARDELL-OLIVER, R., SMETTEM, K., KRANZ, M., AND MAYER, K. Field testing a wireless sensor network for reactive environmental monitoring. In *International Conference on Intelligent Sensors, Sensor Networks and Information Processing* (Dec. 2004), ISNIP 2004, pp. 7–12.
- [12] CAREY, M. J., LIVNY, M., AND JAUHARI, R. The HiPAC project: combining active databases and timing constraints. *ACM SIGMOD Record* 17, 1 (1988), 51–70.
- [13] CHAKRAVARTHY, S., AND MISHRA, D. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.* 14, 1 (1994), 1–26.
- [14] CHEONG, E., LIEBMAN, J., LIU, J., AND ZHAO, F. TinyGALS: A Programming Model for Event-Driven Embedded Systems. In *Proceedings of the 18th Annual ACM Symposium on Applied Computing* (March 2003), SAC’03, pp. 9–12.
- [15] COHEN, J. Non-Deterministic Algorithms. *ACM Comput. Surv.* 11, 2 (1979), 79–94.
- [16] COLMERAUER, A., AND ROUSSEL, P. The birth of Prolog. In *The second ACM SIGPLAN conference on History of programming languages* (Cambridge, MA, 1993), pp. 37–52.
- [17] DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *Journal of the ACM* 7, 3 (July 1960), 201–215.
- [18] DE IPIÑA, D. L., AND KATSIRI, E. An ECA Rule-Matching Service for Simpler Development of Reactive Applications. *IEEE DSONline* 2, 7 (2001). On-line article, Retrieved from <http://dsonline.computer.org/0107/features/lop0107.htm>.
- [19] DOOLIN, D. M., AND SITAR, N. Wireless sensors for wildfire monitoring. In *Proceedings of SPIE Symposium on Smart Structures and Materials* (San Diego, California, USA, Mar. 2005), NDE 2005, pp. 477–484.
- [20] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

- [21] GATZIU, S., AND DITTRICH, K. R. Events in an Active Object-Oriented Database System. In *Proc. 1st Intl. Workshop on Rules in Database Systems* (Edinburgh, UK, Sept. 1993), RIDS, Springer-Verlag, Workshops in Computing, pp. 23–39.
- [22] GAY, D., HONG, W., LEVIS, P., AND POLASTRE, J. Standard TinyOS Sensorboard Interface V1.1. Retrieved from CVS repository [cvs.sourceforge.net/tinyos/tinyos-1.x/docs/spec/sensors.pdf](http://cvs.sourceforge.net/tinyos/tinyos-1.x/docs/spec/sensors.pdf), Feb 2004.
- [23] GAY, D., LEVIS, P., AND CULLER, D. Software Design Patterns for TinyOS. In *Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, June 2005), LCTES 2005, pp. 40–49.
- [24] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The NesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation* (June 2003), PLDI 2003, pp. 1–11.
- [25] GEHANI, N. H., JAGADISH, H. V., AND SHMUELI, O. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases* (San Francisco, CA, USA, Aug. 1992), VLDB 1992, pp. 327–338.
- [26] GLASER, S. D. Some real-world applications of wireless sensor nodes. In *SPIE Symposium on Smart Structures and Materials* (San Diego, California, USA, Mar. 2004), NDE 2004.
- [27] HELLERSTEIN, J. M., HONG, W., MADDEN, S., AND STANEK, K. Beyond Average: Towards Sophisticated Sensing with Queries. In *2nd International Workshop on Information Processing in Sensor Networks* (March 2003), IPSN '03, pp. 3–79.
- [28] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Proceedings of ASPLOS* (Boston, MA, November 2000), NSDI 2004, pp. 93–104.
- [29] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking* (2000), pp. 56–67.
- [30] JAIKAE0, C., SRISATHAPORNPHAT, C., AND SHEN, C.-C. Querying and Tasking in Sensor Networks. In *SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V)* (Orlando, Florida, April 24–28 2000).

- [31] LEE, E. A. What's Ahead for Embedded Software? *Computer* 33, 9 (September 2000), 18–26.
- [32] LEVIS, P. Maté Manual. User Reference Manual, Retrieved from <http://www.cs.berkeley.edu/~pal/mate-web/pubs.html>, 2004.
- [33] LEVIS, P. The Tinscript language. User Reference Manual, Retrieved from <http://www.cs.berkeley.edu/~pal/mate-web/pubs.html>, 2004.
- [34] LEVIS, P., GAY, D., AND CULLER, D. Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines. *In submission (tech report pending)* (May 2004).
- [35] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems* (Berkeley, California, USA, 2003), SenSys 2003, pp. 126–137.
- [36] LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E., AND CULLER, D. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation* (2004), NSDI 2004, pp. 1–14.
- [37] LI, S., LIN, Y., SON, S. H., STANKOVIC, J. A., AND WEI, Y. Event Detection Using Data Service Middleware in Distributed Sensor Networks. *special issue on Wireless Sensor Networks of Telecommunications Systems* 26 (June 2004), 351–368.
- [38] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.* 36, SI (2002), 131–146.
- [39] MAINLAND, G., KANG, L., LAHAIE, S., PARKES, D. C., AND WELSH, M. Using Virtual Markets to Program Global Behaviour in Sensor Networks. In *Proceedings of the 11th ACM SIGOPS European Workshop* (Leuven, Belgium, September 2004).
- [40] MIHAELI, J., AND ETZION, O. Event database processing. In *ADBIS (Local Proceedings)* (2004).
- [41] OUSTERHOUT, J. K. Scripting: higher level programming for the 21st Century. *Computer* 31, 3 (March 1998), 23–30.

- [42] POLASTRE, J., SZEWCZYK, R., AND CULLER, D. Telos: Enabling Ultra-Low Power Wireless Research. In *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors* (UCLA, Los Angeles, California, USA, Apr. 2005), IPSN/SPOTS 2005.
- [43] PRIESTLEY, H., AND WARD, M. A Multipurpose Backtracking Algorithm. *Journal of Symbolic Computation* 18 (July 1994), 1–40.
- [44] RÖMER, K. Determination of Time and Location in Large-Scale Dynamic Networks of Tiny Sensors. In *Advances in Pervasive Computing* (Vienna, Austria, Apr. 2004), A. Ferscha, H. Hoertner, and G. Kotsis, Eds., PERVASIVE 2004, Austrian Computer Society (OCG), pp. 125–132.
- [45] RÖMER, K., AND MATTERN, F. Event-Based Systems for Detecting Real-World States with Sensor Networks: A Critical Analysis. In *DEST Workshop on Signal Processing in Sensor Networks at ISSNIP* (Melbourne, Australia, Dec. 2004), pp. 389–395.
- [46] SERUGHETT, M. OSEK: A super small kernel for deeply embedded applications? *Real-Time Magazine* (1999).
- [47] SIKKA, P., CORKE, P., AND OVERS, L. Wireless Sensor Devices for Animal Tracking and Control. In *29th Annual IEEE International Conference on Local Computer Networks* (Tampa, Florida, USA, Nov. 2004), LCN 2004, pp. 446–454.
- [48] SRISATHAPORNPHAT, C., JAIKAE0, C., AND SHEN, C.-C. Sensor Information Networking Architecture. In *2000 International Workshop on Parallel Processing* (Toronto, Canada, August 2000), ICPP 2000, pp. 23–30.
- [49] STANLEY-MARBELL, P., AND IFTODE, L. Scylla : A Smart Virtual Machine for Mobile Embedded Systems. In *3rd IEEE Workshop on Mobile Computing Systems and Applications* (Dec 2000), pp. 41–50.
- [50] SZEWCZYK, R., POLASTRE, J., AND MAINWARING, A. Lessons from a Sensor Network Expedition. In *1st European Workshop on Wireless Sensor Networks* (Jan. 2004), EWSN 2004, pp. 19–21.
- [51] TITZER, B. L., AND PALSBERG, J. Nonintrusive Precision Instrumentation of Microcontroller Software. In *ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, USA, 2005), LCTES 2005.

- [52] VINOSKI, S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine* (February 1997), 1–12.
- [53] WANG, D., ARENS, E., WEBSTER, T., AND SHI, M. How the Number and Placement of Sensors Controlling Room Air Distribution Systems Affect Energy Use and Comfort. In *International Conference for Enhanced Building Operations* (Richardson, Texas, USA, Oct. 2002), ICEBO 2002.
- [54] WELSH, M., AND MAINLAND, G. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation* (March 2004), NSDI '04, pp. 29–42.
- [55] ZIMMER, D., AND UNLAND, R. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering* (1999), IEEE Computer Society Press, pp. 392–399.

## APPENDIX A

# Original Honours Proposal

**Title:** Rule-Based Programming of Wireless Sensor Networks

**Author:** Mark Kranz

**Supervisor:** Dr Rachel Cardell-Oliver

### A.1 Background

Recent advances in computer hardware technology and radio devices has led to the development of tiny radio devices called motes. These devices are multi-purpose — able to be loaded with different sensor capabilities, are fully programmable, and are able to run on batteries for an extended period of time. Deployed in groups, they are able to use radio communication to establish a local network, which is suitable for sharing sensor data, or collective data processing.

However programming embedded sensor network devices can be a daunting task, even for experienced programmers. We suffer traditional problems with embedded systems, such as concurrency and synchronization issues. There are also a number of radio communication and networking problems, such as collisions and the hidden-terminal problem. We also suffer from extreme constraints, such as limited memory, processing power, and battery power.

To better use the limited resources on a mote, and to address typical embedded system issues, TinyOS [28] was developed. TinyOS uses an extension of C, called NesC, to retain good performance and code size, while adding features such as event models and synchronous code execution. It also contains an extensive library of tools, including network protocols, distributed services, sensor drivers, and data acquisition tools.

An unfortunate issue with TinyOS is a result of the open-source development using a C-based language — there is little consistency from one code module to the next. This makes a steep learning curve for programmers new to sensor

networks, and can produce a number of compatibility issues between different code. To this end, there have been several ways suggested to improve the ease of TinyOS mote programming.

**Standards** Coding standards [22] reduces code conflicts and increases code readability and portability. However, standards must be rigorously defined and followed closely by developers to be useful.

**Application Frameworks** Applications for some well defined purpose, that may be extended to include new goals and requirements. TinyDB [27] uses a query language to allow users to explicitly read samples from motes. This approach is easy from a user point of view, but some new requirements may be impossible given the constraints of the original application.

**Language Extension** TinyGALS [14] is an extension of the standard NesC compiler, to allow programmers to express additional coding constraints. While this can be useful, it only addresses specific difficulties, and still has steep learning curve.

**Interpreters** Maté [34] is a Virtual-Machine (or opcode interpreter) framework, designed to provide easy programming and reprogramming of networks. It addresses performance issues by allowing the machine to be tailored to the application, and interfaces directly with low-level code. It is a good, extensible framework for both beginners and advanced users — but may prove difficult to express overall behaviour of the network. Sensorware [10] addresses the need to express behaviour by using global network scripts, although it does require users to learn a sophisticated scripting language.

**Simulators** TOSSIM [35] is a simulator that can simulate the execution and communication of an entire network. While useful for planning and debugging, its usefulness is limited by the accuracy of the individual models for sensors, radio, and threading.

## A.2 Aim

### A.2.1 Hypothesis

“To address the shortcomings of these approaches, we propose a different embedded system model. We propose that a *rule-driven interpreter* will provide:”

- *Concise* expression of local and global behaviour

- *Flexibility* to changing behaviour requirements
- A *framework* for developing environment aware reactive applications
- A *robust* execution model

### A.2.2 Justification

Rule-driven approaches have been explored in a number of different languages, systems and applications over the last few decades. From 1st order logic [16] to Event-Condition-Action (ECA) databases [18], rules give systems the power to conditionally react to internal and external events. They are typically more intuitive than traditional imperative languages, and are often capable of being expressed declaratively — describing *what* to do, but not *how* to do it. This sort of approach would be a significant step in simplifying the difficult task of specifying the behaviour of embedded sensor networks.

However, when coupled with a high-level interpreter, it would also aid users to alter or extend the behaviour of these systems. An approach in which rules are interpreted and executed at run-time, rather than at compile time, would reduce the arduous task of reprogramming, testing, and maintaining deployed networks. Rules would be compact if expressed in an interpreted language, and could thus be loaded and registered via radio uplink. SensorWare [10] demonstrates that it is advantageous for program code to implicitly migrate between sensor nodes. Maté [34] has shown that an interpreter “tailored to a particular deployment can provide re tasking flexibility, while retaining program efficiency”. A rule based interpreter should aim to follow these approaches.

Robustness of execution for sensor-network applications requires consideration of a number of factors. The primary requirement in this case shall be error-detection and error-recovery for buggy code — ensuring that poor user code will not irrevocably crash the system. We shall also consider the continued survival of the network to be a factor of robustness, and thus we will consider minimising power-usage and battery lifetime a critical requirement. We thus propose that a run-time interpreter will provide effective error and power management, through abstraction from application-level code.

### A.2.3 Goals and Success factors

This hypothesis will be tested by attempting to meet a set of goals:

1. Repeat the functionality of an existing reactive embedded system. Express the system's behaviour as a set of rules, and implement using simple condition-action model.
2. Change this system's behaviour to meet new requirements. Modify the rule-set to make these behaviour changes.
3. Assess robustness features required for long-term deployment, including
  - Battery usage
  - Radio and Network communication
  - Software error

#### A.2.4 Anticipated Accomplishments and Deliverables

In meeting the goals of this project, we expect to deliver the following:

**Interpreter** Implementation of an ECA rule interpreter for sensor networks

**Debugging and diagnostic tools** Tools for assessing networks<sup>1</sup>

**Framework** Guidelines and procedures for adding support for new platforms and sensors

**Test cases** A set of general and specific test cases, suitable for any embedded system interpreter

**Software Assessment** Testing and reporting defects in TinyOS and NesC to the open-source community

<sup>1</sup> *Debugging tools may need to be developed for the purpose of this project, since there is a lack of general purpose tools in this area. Such tools should be reusable for other environments.*

### A.3 Method

Due to limitations on time for this project, it may not be feasible to implement a complete interpreter from scratch. Therefore, we have decided that it would be appropriate to spend time assessing the suitability of using an existing interpreter [34], and introducing the ECA architecture. Even if we pursue the

development of a new ECA interpreter, the techniques employed by Maté [34] will give us valuable insight.

Regardless of the alternative chosen, substantial time must be allocated towards both the development of the interpreter, and sample applications to run with it. Time must also be allowed to perform system and user tests, since these will be the factors determining the success of the project.

To meet the (ambitious?) goals of this project, we have decided to use an incremental approach, where we will aim to produce small achievements early, and build upwards. Lower priority ‘wish-list’ items will be designated towards the end of the project, only to be attempted if there is sufficient time remaining.

Our initial timeline<sup>1</sup> is outlined in Table A.1.

<sup>1</sup> *Tasks before December have additional time allocated, due to other study commitments*

## A.4 Software and Hardware Requirements

During recent TinyOS work, we have customized a working environment with all required software and hardware. The list below merely mentions what tools are involved, *not* what needs to be acquired.

### A.4.1 Required Software

**TinyOS** Requires Linux, or Windows running Cygwin

**TinyOS development tools** Including customized text editor, CVS tools, serial port software (uisp)

**Java** jdk1.4.2 or higher

**CASE tools** Basic package modeling, such as ArgoUML or GraphViz (TBD)

### A.4.2 Required Hardware

**Workstation** Suitable x86 computer, (2Ghz equivalent or higher), 2 available serial ports.

**Cables** 2x Serial COM cable or USB to serial converter

**Mote** 5x Crossbow Mica2 433 MHz motes

Task	Task	Completion Date
Further Research ECA systems Maté interpreter	5%	August
Design ECA architecture Test applications	20%	October
Setup and Preparation TinyOS setup Test stubs and makefile Versioning	5%	October
Debugging Tools Design, examine existing Implement	10%	November
Condition Engine Event Controller Condition Loader Condition Engine	15%	January
Action Engine Simple Action model Concurrent Action Engine	15%	February
System Tests Complete ECA set Concurrent ECA sets Test Application	15%	March
Libraries General purpose Deployment specific	5%	March
Assessments Thesis Seminar Poster	10%	April

Table A.1: Project Timeline and Work

**Mote interface-board** 2x Crossbow MIB510 programmers

**Sensor-board** 2x Crossbow MDA300CA general purpose sensor-boards

**Analog sensors** 1+ Analog Sensor (Decagon ECHO-20 soil-moisture sensor)

**Digital sensors** 1+ Digital Sensor (Decagon Rain Gauge)

**Batteries** 15 NiMH batteries

## APPENDIX B

# Mote Hardware Comparisons

Mote Type	WeC 1998	Rene 1999	Dot 2000	Mica 2001	Mica2Dot 2002	Mica 2 2002	Telos 2004
<b>Microcontroller</b>							
Type	AT90LS8535	ATmega163	ATmega163	ATmega128	ATmega128	ATmega128	TI MSP430
Program memory (KB)	8	16	16	128	128	128	60
RAM (KB)	0.5	1	1	4	4	4	2
Active Power (mW)	15	15	15	8	8	33	3
Sleep Power ( $\mu$ W)	45	45	45	75	75	75	6
Wakeup Time ( $\mu$ s)	1000	36	36	180	180	180	6
<b>Nonvolatile storage</b>							
Chip	24LC256	24LC256	24LC256	AT45DB041B	AT45DB041B	AT45DB041B	STM24M01S
Connection type	I2C	I2C	I2C	SPI	SPI	SPI	I2C
Size (KB)	32	32	32	512	512	512	128
<b>Communication</b>							
Radio	TR1000	TR1000	TR1000	TR1000	CC1000	CC1000	CC2420
Data rate (kbps)	10	10	40	40	38.4	38.4	250
Modulation type	OOK	OOK	ASK	ASK	FSK	FSK	O-QPSK
Receive Power (mW)	9	9	12	12	29	29	38
Transmit Power (mW)	36	36	36	36	42	42	35
<b>Power Consumption</b>							
Minimum Operation (V)	2.7	2.7	2.7	2.7	2.7	2.7	1.8
Total Active Power (mW)	24	24	27	27	44	89	41
<b>Prog Interface</b>							
Expansion	none	51-pin	none	51-pin	19-pin	51-pin	10-pin
Communication	RS232	RS232	RS232	RS232	RS232	RS232	USB
Integrated Sensors	no	no	yes	no	no	no	yes

Figure B.1: The family of motes and their capabilities [1]

## APPENDIX C

# Communication Abstraction

This appendix contains a short summary of the communication abstractions briefly mentioned in Section 2.4. While not essential to our discussion of event-detection, it is nevertheless an interesting topic, reducing the burden of handling distributed algorithms and communication within sensor networks.

### C.0.3 Directed Diffusion

Directed Diffusion is a data-delivery protocol that operates in a similar manner to database oriented networks, whereby data flow is governed by the actions of producers and consumers. It aims to abstract communication primitives into a tree-based routing protocol, and uses named attributes to produce data-centric applications. Tasks are specified according to temporal and spatial constraints, such as intervals, location and attribute types. By making tasking and routing decisions locally, directed diffusion is suitable for scalable, macro-programmed networks.

Directed Diffusion ties the routing and communication protocols to the network behaviour — while this may provide application-level energy optimizations, it may also limit network behaviour. As with other macro-programming approaches, it becomes difficult to express reactive applications, and is unsuited to unmaintained, self-organising networks. Even when applications are limited to data gathering, it has been found that maintaining a routing tree compromises energy of the network.

### C.0.4 Abstract Regions

Abstract Regions is a communication abstraction that provides a family of spatial operators to capture local communication within regions of the network. Its primary aim is to provide a set of interfaces that supports identification of neighbours, data sharing, and data compression, to be used in higher-level languages

and application programs.

While networks have more severe constraints than typical parallel systems, inspiration can be drawn from parallel computing tools [54]. Abstract Regions is likened to Message Passing Interface (MPI), a parallel computing paradigm that shields programmers from machine and communication specifics, yet remains powerful and general enough to allow a range of applications and application-specific optimizations. Such abstractions are not yet adopted by the sensor-network community, as there has been little consensus or agreement on standards. Subsequently, time spent building applications, even with the support of the tools and frameworks we are discussing, is still dominated by the development of routing, data-collecting and energy management components.

The key concept behind abstract regions is the formation of different communication regions. These regions not only form the backbone of the routing tree, but are the basis of data-sharing and aggregation operations and allow the tuning the network according to energy and communication requirements. Region are constructed based on spatial attributes, such as geographic location and radio connectivity.

A sensor node with abstract region capabilities can perform the following operations.

**Neighbour discovery** : This consists of gathering the required information for discovering the nodes neighbours within the region, for example, using GPS to retrieve the location, or estimating radio link quality. This process may run continuously, or be started and stopped as required.

**Enumeration** : Allows a node to find all nodes participating in the region.

**Data sharing** : Data is stored as named key/value variables. Nodes may either add/modify local variable, or request to retrieve a variable from another node.

**Reduction** : Data reduction, (compression, aggregation) allows nodes to reduce a shared variable across the network, according to a reduction operator (min, max, avg). The result is stored in a globally accessible shared variable.

These operations gives programmers a rich API with which to construct WSN applications. While most implementation details are hidden from the programmer, Abstract Regions can also expose a set of tunable parameters to provide better control. Features such as performance feedback, and energy/latency tradeoffs gives applications control over the low level behaviour of the Regions protocols.

Abstract Regions is currently supported on TinyOS platforms, and uses a NesC compiler extension SplitNesC. While it is a good abstraction, it is merely a concurrency primitive that supplies the building blocks for higher-level languages — it will be some time before the tools to support Region’s programming are mature enough to build complex and varied systems.

### C.0.5 Virtual Markets

Virtual Markets, or Market-based Macroprogramming (MBM) is a sensor network paradigm modelled on an virtual economic market — a technique developed for decentralised optimisation. Nodes achieve their local goals through trading *goods* such as using sensors or transmitting data, based on the global price of network resources. Global efficiency is achieved through agent-utility optimisation, subject to energy and communication constraints. The network behaviour can be tuned by adjusting the prices for node actions, stimulating the node-agents to react and optimise accordingly.

It is an interesting concept, since it allows the construction of completely independent, self-managing networks, and has the potential to ‘train’ networks to adopt complex behaviour subject to strong and/or variable constraints. In addition, users can interact with the network by assuming control of one or more network nodes, and adjusting economic variables such as price and demand.

This approach shows promise in areas where direct specification of network behaviour is difficult, however, it is yet to be seen whether it is entirely suitable for well-known, simple tasks. For instance, a user may wish to retrieve a single set of data from a self-organised MBM network — to do this through adjusting market prices is not necessarily logical or straightforward. Perhaps the development of additional support tools and scripts would allow MBM networks to handle both abstract requirements and direct queries, providing a balance between data-centric and control-centric strategies.

## APPENDIX D

# Maté Traffic Example

<i>Subscription Context</i>	
<pre>begin(ATMOST(50), IMMEDIATE)  init(FALSE, car_e(), ADD) op(FALSE, car_e(), FIRST) term(FALSE, MINUTE, TIMER) filter(1, DATA, LEQ, limit-20)  jam_s = load()</pre>	<pre>begin(ATLEAST(50), IMMEDIATE)  init(FALSE, car_e(), ADD) op(FALSE, car_e(), ADD) term(FALSE, MINUTE, TIMER) filter(1, DATA, GEQ, limit-10)  cleared_s = load()</pre>
<pre>begin(SEQUENCE, IMMEDIATE)  init(FALSE, accident_e(), ADD) op(TRUE, jam_s, FIRST) term(FALSE, 15*MINUTE, TIMER)  accident_s = load()</pre>	<pre>begin(UNLESS, DEFERRED)  init(TRUE, accident_s, ADD) op(FALSE, rerouted_e(), FIRST) op(TRUE, cleared_s, FIRST) term(FALSE, 15*MINUTE, TIMER)  reroute_worked_s = load()</pre>
<i>Action2 Context — Reroute</i>	
<pre>private location = getparticipants()[0] closelane(location) reroute() ...</pre>	
<i>Action3 Context — Dispatch Team</i>	
<pre>private location = getparticipants()[0] dispatchteam(location) ...</pre>	