Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-ingenieurswetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

Language Support for Programming Interactions among Wireless Sensor Network Nodes

Proefschrift ingediend met het oog op het behalen
van de graad van Master in de Ingenieurswetenschappen: Computerwetenschappen

Wouter Amerijckx

Promotor: Prof. Dr. Wolfgang De Meuter
Begeleiders: Dr. Coen De Roover
Christophe Scholliers

JUNI 2011

Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Language Support for Programming Interactions among Wireless Sensor Network Nodes

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Master of Engineering: Computer Science

Wouter Amerijckx

Promotor: Prof. Dr. Wolfgang De Meuter
Advisors: Dr. Coen De Roover
Christophe Scholliers

JUNE 2011

Acknowledgements

This dissertation would not have been what it is today without the outstanding support of various people.

First of all, I'd like to thank Professor Wolfgang De Meuter for promoting this thesis and giving me the opportunity to conduct my apprenticeship at the Software Languages Lab.

In addition, I'd like to thank my advisors dr. Coen De Roover and Christophe Scholliers for all their support and time spent while keeping me on track. The input they've given during hours of discussions and after proofreading this thesis was invaluable for the final result. Even when they were overwhelmed by their own work, they found the time to give me advice, motivate me and help me out when I got stuck.

Next to my advisors, I would also like to thank all other members of the Software Languages Lab for their interest and input during the various presentations about my work. Additionally, I'd like to thank the members of the Stadium project for inviting me to their meetings and workshops. They introduced me to the LooCI middleware, which lies at the basis of the work presented in this thesis.

Last but not least, I'd like to thank my mother and friends for supporting me during my education. In particular, for the many talks, laughs and encouragements at the times when I needed them most.

Samenvatting

De vooruitgang in hardware technologie heeft ertoe geleid dat computers krachtiger, kleiner, en goedkoper worden. Samen met de technologische vooruitgang in draadloze netwerken heeft ze aanleiding gegeven tot een nieuwe vorm van gedistribueerde systemen, de zogenaamde draadloze sensor netwerken (DSN). DSN bestaan uit minuscule toestellen op batterijen die draadloos communiceren. Deze toestellen zijn goedkoop, uitgerust met sensoren en actuatoren, en breed inzetbaar. Tot op heden werden DSN reeds gebruikt voor het inspecteren van zebra's en het besturen van aircoconditioning systemen. Een nieuwe trend bestaat erin DSN te gebruiken voor *actieve toepassingen* waarbij het DSN moet reageren op de gemeten sensorwaarden. Ondanks deze trend bieden de huidige programmeermodellen voor DSN niet de nodige abstracties om het ontwikkelen van deze toepassingen te vergemakkelijken.

Het ontwikkelen van actieve toepassingen houdt normaliter in dat de individuele DSN toestellen, typisch *knopen* genoemd, één voor één geprogrammeerd dienen te worden. Voor elke knoop moet de ontwikkelaar bepalen hoe die interageert met andere knopen in het netwerk. In deze interacties worden berichten uitgewisseld en verwerkt om de nodige acties te starten. Deze berichten bevatten meestal sensorwaarden en kunnen bijvoorbeeld verwerkt worden om een actuator aan te sturen (bvb., een radiator). De huidige programmeermodellen hebben het versturen van deze berichten wel vergemakkelijkt, maar de complexiteit van het verwerken van binnenkomende berichten werd door de meesten genegeerd. Dit deel van de interactie dient typisch geïmplementeerd te worden in een *event handler* die aangeroepen wordt bij het ontvangen van elk nieuw bericht. Het is echter reeds aangetoond dat het gebruik van event handlers in strijd is met verscheidene software engineering principes zoals samenstelbaarheid en schaalbaarheid. Een event handler kan niet zomaar samengevoegd worden met een andere om bijkomende berichten te verwerken. Bovendien wordt een event handler erg complex indien die moet reageren op een sequentie van verscheidene berichten die op verschillende ogenblikken ontvangen worden. Dit houdt in dat bijgehouden moet worden welke berichten al dan niet reeds ontvangen werden. Ad hoc oplossingen voor het verwerken van binnenkomende berichten zijn foutgevoelig en dienen daarenboven gedupliceerd te worden in de event handlers van verschillende knopen.

Deze verhandeling introduceert ondersteuning voor het programmeren van interacties tussen knopen om de hierboven vermelde problemen het hoofd te bieden. We introduceren een domein-specifieke programmeertaal die gebruikt dient te worden bovenop event-gebaseerde middleware. Deze taal laat toe om de interacties van een knoop met anderen uit te drukken via *declaratieve regels* en vermijdt op deze manier de complexiteit van event handlers. Daarbovenop introduceren we een macroprogrammeertaal die toelaat om toepassingen voor DSN in hun geheel te programmeren. Deze taal bevordert het hergebruik van programmacode binnen en tussen verscheidene toepassingen. Onze benadering minimaliseert de toevallige complexiteit eigen aan het ontwikkelen van DSN toepassingen met event-gebaseerde middleware en stelt de ontwikkelaar in staat zich te focussen op de essentiële complexiteit van de toepassing.

Abstract

Advances in hardware technology have lead to computing devices becoming smaller, cheaper and more powerful. Together with the technological advances in wireless networks, they gave rise to a new form of distributed systems, called wireless sensor networks (WSNs). WSNs consist of tiny, sensor- and actuator-equipped, battery-powered devices that communicate wirelessly. Due to the low cost of these devices, WSNs can be widely deployed. For instance, they have already been deployed for monitoring zebras and controlling air-conditioning systems. An emerging breed of applications, called *sense-and-react applications*, tasks a WSN not only with sensing, but also with reacting to sensor readings. However, current programming models for WSNs do not offer appropriate abstractions to deal with the complexity of implementing these applications.

Developing sense-and-react applications typically involves programming individual WSN devices, often called *nodes*, one by one. For each node, the developer has to specify how it interacts with the other nodes in the network. In these interactions, messages are exchanged and processed to trigger the appropriate actions. These messages typically carry sensor data and can, for instance, be reacted to by controlling an actuator (e.g., a heater). While existing programming models have facilitated transmitting messages, most of them have ignored the difficulties of processing incoming messages. For this part of the interaction, they force the developer to implement the required logic in an event handler that will be invoked whenever a new message is received. However, the use of event handlers has already been shown to violate a range of software engineering principles including composability, scalability and separation of concerns. For instance, an event handler cannot simply be composed with another one to have it process additional messages. In addition, reacting to a sequence of time-dispersed messages complicates an event handler as it implies keeping track of what messages have already been received. Ad-hoc solutions for processing incoming messages are error-prone and bound to be duplicated over the event handlers of multiple nodes.

This dissertation introduces language support for programming interactions among WSN nodes to address the aforementioned problems. Concretely, we introduce a new domain-specific programming language to be used on top of event-based WSN middleware. This language enables expressing the interactions of a node with others through *declarative rules* and thereby avoids the complexity of event handlers. In addition, we introduce a macroprogramming language for programming WSN applications as a whole. This language fosters code reuse within and among WSN applications. Our approach minimizes the accidental complexity inherent to programming WSN applications using event-based middleware and enables the developer to focus on the application's essential complexity instead.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Advances in hardware technology have lead to computing devices becoming smaller, cheaper and more powerful. Together with the technological advances in wireless networks, they gave rise to a new form of distributed systems, called wireless sensor networks (WSNs). WSNs consist of tiny, sensor- and actuator-equipped, programmable devices that communicate wirelessly. These devices, often called *nodes*, have highly constrained computing- and communication resources and are typically battery-powered. Even though hardware advances play an important role in the development of WSN technology, the most important factor is the programming model supporting application development [1]. Unlike traditional distributed systems, WSNs have a *dynamic* nature: at any time, nodes can fail or get disconnected from the network (i.e., disconnections are the norm rather than an exception). Due to this dynamic nature, traditional programming models could not cope with the requirements of WSNs and a new body of programming models was introduced by the academic community.

Two major approaches can be taken when programming WSN applications. The first approach consists of programming every individual node by specifying its interactions with the other nodes. These interactions typically involve sending data around, storing data, or reacting to requests either by actuating, or by sensing and answering the sensed values. This approach is commonly referred to as *node-centric programming* [2]. While node-centric programming models give the developer fine-grained control over important requirements such as a node's power- and bandwidth consumption, they also require the developer to deal with low-level details.

The second approach allows the programmer to take distance from low-level details and consists of programming all nodes in the WSN as a whole. This approach is commonly referred to as *network-centric programming* [2]. Network-centric programming models either abstract the WSN as a whole or allow the WSN to be programmed from a macroscopic viewpoint. For instance, a common abstraction represents a WSN as a virtual database of sensors that can be queried using an SQL-like language [3], while a common way to allow

macroscopic programming consists of supporting mobile code [4]. Of course, using higher level programming models comes at the price of lesser control over the individual nodes.

To date, WSNs have been used mostly for sense-only applications. Examples are applications for monitoring habitats [5], zebras [6] and glaciers [7], and applications for detecting intrusions [8], forest fires [9] and flooding rivers [10]. Only recently, WSN applications became more active. An example is an application for controlling heating- and air conditioning systems [11]. Especially the development of active applications, typically referred to as *sense-and-react* applications, remains difficult as this requires a high degree of control over the individual nodes (e.g., to steer actuators).

The following sections show the problem that appears when programming sense-and-react WSN applications and how our work tackles this problem. To conclude the introduction, we present a high-level overview of this dissertation.

1.1 Problem Statement

For developing sense-and-react applications, a developer typically relies on a node-centric programming model to program node-level code. This node-level code has to process sensor-data obtained through interactions with other nodes and based on this processing, reactions have to be triggered (e.g., steering an actuator). However, contemporary node-centric programming models do not offer adequate support for programming node interactions.

Node-centric programming models require the developer to program at a low level of abstraction. For instance, programming an interaction with another node typically involves encapsulating data in a packet and transmitting this packet in the node's one-hop neighborhood. Processing data upon reception, on the other hand, often involves the use of an event handler in which received packets have to be parsed and their data has to be extracted. To raise the level of abstraction, there exist several solutions that complement node-centric programming models with additional support for programming interactions. However, while these solutions facilitate transmitting data, most of them do not help in processing data upon reception. They still rely on the use of event handlers for this part of the interaction. Not only do event handlers complicate programming interactions among sensor nodes, they also force the programmer to adopt a programming style that violates several software engineering principles, including composability, scalability and separation of concerns [12].

To illustrate how event handlers complicate programming interactions among sensor nodes, we can consider the programming model introduced by event-based WSN middleware. Such midleware brings the Publish-Subscribe interaction pattern [13] to WSNs and allows sensor nodes to communicate by exchanging events via a decentralized event bus. Nodes can *publish* events of certain types, which will then be routed to other nodes that *subscribed* to these events' types. Published events are usually only routed to subscribers that are online at the time of publication. While Publish-Subscribe decouples the communication between sensor nodes (i.e., the communicating parties don't have to be fixed at compile-time), it still requires the processing of received events to be implemented in an event handler. Therein, a developer typically dispatches over the possible event-types that could be received and selects the appropriate reaction.

An example is given in Listing 1.1.

```
1 public void receive(Event ev) {
2   if(ev.getType() == EventTypes.GET_TEMP)
3     publish(new Event(EventTypes.TEMP, getTemperature()))
4 }
```

Listing 1.1: A simple event handler

The example shows a simple event handler that reacts to one particular event-type: when a temperature request is received, the current temperature is measured and published. Typically, however, the event handler becomes larger and more complex as *more event-types* have to be reacted to. Extending an event handler by composing it with another one typically doesn't work. Furthermore, it's not hard to imagine cases in which it might be desired to react to *several related but time-dispersed events*. One example is a sensor node steering heaters throughout a house. To steer a heater in a particular room, it is required to combine sensor-reading events published by temperature- and humidity sensors in the same room. Implementing this behavior further complicates the event handler. It requires additional bookkeeping to keep track of which events have already been received, additional matching of related events (i.e., events originating from sensors in the same room), and additional state management to make sure no stale events are used for steering the heaters. While in this example, events are only related by the room they originate from, more sophisticated relations require even more sophisticated event handler implementations (e.g., specifying a finite state machine that transitions between states upon the reception of events).

Without adequate support for processing received data, developers have to deal with the requirements mentioned in the above example by designing ad-hoc solutions that are bound to be duplicated over the event handlers of multiple nodes. Worse, a sensor node's application logic would be polluted with the code for dealing with these requirements. As a consequence, the application logic becomes difficult to understand, to extend, and to reuse. While this problem plagues other event-driven architectures as well, existing solutions like complex event processing do not readily translate to the setting of wireless sensor networks as they are not designed for use in distributed systems.

1.2 Proposed Solution

To tackle the aforementioned problem, we propose to introduce additional language support for programming interactions among WSN nodes. Concretely, we propose both a node-centric- and a network-centric programming language.

The node-centric programming language constitutes a building block, to be used on top of existing event-based WSN middleware. It allows the use of *declarative rules* to specify the interactions between sensor nodes. This declarative model addresses the various problems that using an event handler for programming node interactions brings along. The language reduces the complexity and improves the modifiability and scalability of the application logic for WSN nodes. Its runtime employs the well-known Rete algorithm for the evaluation of interaction rules.

The network-centric programming language is a macroprogramming language that allows each node's behavior to be specified in a centralized fashion.

This language supports macro-definitions and import statements, thereby fostering code reuse when building WSN applications. In addition, it facilitates developing WSN applications as the interactions between the sensor nodes become clearly visible.

1.3 Document Overview

This dissertation introduces a new domain-specific programming language for developing wireless sensor network applications from a node-level perspective and a complementary macroprogramming language that enables development from a network-level perspective.

Chapter 2 gives a non-exhaustive overview of the state of the art for programming wireless sensor networks. Existing solutions are covered by first focussing on programming models that support the development of traditional, sense-only WSN applications and afterwards focussing on programming models that support the development of the more recent, sense-and-react applications.

Our main solution, the node-centric language dubbed CrimeSPOT, is presented in Chapter 3. All features are explained by incrementally implementing the running example presented in Section 3.1. More details concerning CrimeSPOT's prototype implementation can be found in Appendix A and a preliminary evaluation of its overhead can be found in Appendix B.

In Chapter 4, the focus moves to Macro-CrimeSPOT, our macroprogramming language that facilitates programming WSN applications using the CrimeSPOT language by moving from a node-level perspective to a network-level perspective. In this chapter, we also revisit the running example and discuss its final implementation.

To conclude, we validate our solutions through the implementation of illustrative example applications in Chapter 5 and discuss our contributions and future work in Chapter 6.

2 State of the Art

This chapter gives a non-exhaustive overview of the state of the art for programming wireless sensor networks (WSNs). To date, several so-called *programming models* have been proposed, which each target different requirements [1, 2]. In this chapter, we will focus on the typical ones, relevant to the problem at hand.

After introducing programming models for traditional WSNs, the focus moves to programming models for wireless sensor- and actuator networks (WSANs). WSANs are WSNs that employ, next to sensors, actuators to control the environment. As we will discuss, programming WSANs requires more control over the individual nodes and to this end, the typical programming models for traditional WSNs are not appropriate.

2.1 Traditional Wireless Sensor Networks

The applications that made wireless sensor networks popular are *sense-only* applications in which sensor data is extracted from the network. Typical examples are applications for environmental monitoring and scientific data gathering [5–7]. Programming models that support the development of these applications abstract the WSN as a single, abstract machine, that can be programmed as such.

The most common abstraction for a WSN is *a virtual database of sensors*, from which raw sensor data can be extracted through SQL-like queries. By using this database abstraction, a developer can easily extract sensor data without having to worry about how it is actually obtained. However, in some cases, it is desired to express more advanced queries for which SQL-like languages are not appropriate. To this end, another type of programming models allows developers to *program queries* through more advanced languages (e.g., functional programming languages). A third typical type of programming models allows a developer to register event definitions to *extract interpreted data* from the network. An event definition specifies an event (e.g., an explosion) by listing its characteristics (i.e., the corresponding measured sensor values). After register-

ing event definitions, the programming model's runtime will monitor the WSN and will inform the end-user when the registered events have been detected.

In the following sections, three programming models that are each offering one of the above functionalities will be discussed in more detail.

2.1.1 Raw Data Extraction

The most straightforward way to extract raw data from a WSN is to make use of a programming model that offers a *database abstraction*. Among the models offering this abstraction are TAG [14], COUGAR [15], SINA [16] and TINYDB [3]. In this section, we will give a brief overview of TINYDB, which is representative for this category of programming models.

In TINYDB, the WSN is abstracted as a large `sensors` table, which has a row for every WSN node for every moment in time, and a column for each data item that can be produced by a node. The programmer can issue queries on this table from a base station, which are then injected in the network. These queries can be expressed in a language very similar to SQL. Listing 2.1 shows an example query, modified from [3]. Note that TINYDB supports aggregates such as `AVG`, `SUM` and `COUNT`, and also allows to define custom aggregates.

```
1 SELECT AVG(volume), room
2 FROM sensors
3 WHERE floor = 10
4 GROUP BY room
5 SAMPLE PERIOD 30s FOR 10 minutes
```

Listing 2.1: TinyDB: example query

Assuming that every WSN node provides a room and a floor value, this query reports the average volume in all rooms on the tenth floor. The query has a lifetime of ten minutes, in which a new update will be delivered to the base station every 30 seconds (i.e., the sampling period). In the absence of an explicit sampling period, TINYDB will automatically decide the sampling according to the query's lifetime and the remaining battery level of the nodes involved in the query's processing. This is only one of the various optimizations that TINYDB uses to be as energy-efficient as possible.

Next to queries for pure data collection, TINYDB also supports *actuator queries*, which will invoke an external command whenever they are satisfied. As an example, consider the query, modified from [3], in Listing 2.2.

```
1 SELECT nodeid, temp
2 FROM sensors
3 WHERE temp > 40
4 OUTPUT ACTION power-on-fan(nodeid)
5 SAMPLE PERIOD 60s
```

Listing 2.2: TinyDB: an actuator query

This query will verify the in-network temperatures every 60 seconds, and will invoke the `power-on-fan` command as soon as a measured temperature exceeded 40 degrees.

As the above examples illustrate, TINYDB allows programmers to express data extraction applications in a very concise notation. While TAG, COUGAR, and SINA offer similar functionality, SINA extends this functionality with the

ability to send *tasks* to sensor nodes. Any task can be expressed in SINA's SQTL language and will be executed on the target nodes. Optionally, a task can also return results to the base station. For instance, a vehicle tracking algorithm can be implemented in SQTL [16].

2.1.2 Programming a Query

As mentioned earlier, the SQL-like languages offered by the programming models that employ a database abstraction can be insufficient to express advanced queries. As an example of an advanced query, consider a query that generates an output whenever a fire is detected. While a fire can be detected by reading out temperature sensors and verifying whether the readings exceed a certain threshold, false positives also have to be filtered out. This can be done, for instance, by verifying whether the nodes in the immediate neighborhood of the nodes detecting a high temperature also measure a high temperature. Clearly, using the database abstraction would require the initiator of the temperature query to filter the results manually. However, programming models like REGIMENT [17] allow queries like this to be programmed in their entirety.

REGIMENT is a functional programming language that is based on the concept of Functional Reactive Programming (FRP). It is designed to compile and install long-running queries, which can be expressed by manipulations of data streams. REGIMENT has two basic concepts: *signals* and *regions*. A *signal* represents a data stream (i.e., a time-varying value), while a *region* represents a collection of multiple signals. For instance, a sensor reading and the result of a node's computation are both signals, as they can vary over time. A region, on the other hand, can represent all measured temperature readings in a given geographical region. Additionally, multiple signals within a region can also be aggregated, resulting in a new signal (e.g., the sum of certain measured temperature readings).

To clarify these concepts, we can consider the example from Listing 2.3 (modified from [17]), which implements the fire detection query discussed above.

```
1 fun readTemp(node) { sense("temp",node) }
2 fun tempTuple(node) { (node, readTemp(node)) }
3 fun aboveTresh((node,temp)) { temp > TEMP_THRESHOLD }
4 fun sumTempRegion(reg) { rfold((+), 0, reg) }
5
6 possibleAlarms = rfilter(aboveTresh, rmap(tempTuple,world));
7 hoods = rmap( fun(n,t) { khood(1,n) }, possibleAlarms);
8 tempHoods = rmap( fun(reg) { rmap(readTemp,reg) }, hoods);
9 tempSums = rmap(sumTempRegion, tempHoods);
10 BASE <- rfilter( fun(ts) { ts > CLUSTER_TEMP_TRESH }, tempSums);
```

Listing 2.3: Regiment: example query

Four auxiliary functions are defined for the query (lines 1-4): to read out the temperature sensor of a particular node (`readTemp`), to place a node's identifier and its measured temperature in a tuple (`tempTuple`), to verify whether the temperature in such a tuple is above a certain threshold (`aboveTresh`), and to sum all temperatures in a region (`sumTempRegion`). Note that signals in a region can be aggregated by using the predefined `rfold` function together with a combination function and an initial value (e.g., + and 0 in this case).

The statements on lines 6 to 10 implement the query. To this end, several regions are defined. `possibleAlarms` represents the region consisting of temperature reading tuples that exceed a certain temperature threshold. This region is obtained by mapping `tempTuple` over the predefined `world` region (i.e., the region containing all nodes in the WSN) and filtering the results. `hoods`, on the other hand, represents a nested region; it contains regions with all nodes of the one-hop neighborhood of the nodes in the `possibleAlarms` region. These one-hop neighborhood regions are obtained by mapping the predefined `khood` function over the `possibleAlarms` region. To filter out false positives, the temperature has to be measured on each node of these sub-regions and the results have to be summed per sub-region (lines 8-9). Finally, fires are detected by verifying whether any of these sums exceed a certain cluster temperature threshold, and the exceeding sums are sent to the base station (line 10).

Even though the REGIMENT language is not as concise as the previously discussed SQL-like languages, it does allow the programmer to program a wide range of queries.

2.1.3 Interpreted Data Extraction

While in the previously discussed programming models, the developer has to issue queries for raw sensor data, other programming models allow the developer to register higher-level *events* to work with. This notion of *event* raises the level of abstraction as it associates semantics with raw sensor data. For instance, it can be specified that an explosion event corresponds to the detection of a high temperature and an intense light. After registering such event definitions, they are interpreted by the programming model's runtime and allow to extract higher-level data from the sensor network. Among the programming models that allow to extract interpreted data from a WSN are SEMANTIC STREAMS [18] and DSWARE [19]. Even though both models are similar and allow the definition of events, SEMANTIC STREAMS requires the end-user to issue queries over these events, while DSWARE notifies the end-user when the registered events have been detected. In this section, we will give a brief overview of DSWARE.

In DSWARE, an event has to be described by a list of sub-events that have to take place in a given time window and geographical region. A sub-event can be either a predicate on an observed sensor reading, or another high-level event. In addition, it can be specified that not all sub-events have to take place for a detection of the high-level event to occur. This can be specified by a *confidence function*, which computes the confidence with which the event took place, and a *minimum confidence* required for the event to be detected. A confidence function can be arbitrarily defined and takes as arguments a boolean for every sub-event. This boolean indicates whether the sub-event took place in the given time window and geographical region.

As an example, consider the definition of the explosion event shown in Listing 2.4, modified from [1]. Note that event definitions are registered in an SQL dialect.

```
1 INSERT INTO EVENT_LIST(explosion, AREA, [0,0;200,200], SUBEVENT_SET,
2                        user_base_station, 1 sec,
3                        1 hour)
```

```
4
5 SUBEVENT_SET (
6         SAFETY_TIMEOUT,
7         (temperature > 60),
8         (light > 200),
9         (compareSound(sound,explosionSignature))
10 )
```

Listing 2.4: DSWare: An Event Definition

Lines 1 and 2 specify that the runtime should monitor for the explosion event in the specified geographical area and report detections to the base station with a maximum delay of one second. In addition, line 3 specifies that this definition remains valid for one hour. Lines 5 to 10 specify the explosion's subevents and the corresponding time window in which they have to take place (i.e., `SAFETY_TIMEOUT`). For simplicity, there's no confidence function, nor minimum confidence level. An explosion is detected as soon as a high temperature, an intense light, and a sound whose signature resembles that of an explosion have been detected. Based on this definition, the WSN is configured by DSWare to meet the QoS requirements and to be as energy-efficient as possible.

2.1.4 Evaluation

Contemporary programming models for WSNs offer high-level and well-suited abstractions to support the development of traditional, sense-only WSN applications. The WSN is abstracted as a single, abstract machine that can be programmed as such. None of the discussed solutions require a programmer to worry about *how* data is obtained from a WSN (e.g., about which nodes have to communicate to propagate the sensor data to the base station). As such, traditional WSN applications can be programmed quite straightforwardly.

However, this very high level of abstraction is not always appropriate. For instance, when building WSANs, a high degree of control is required over the individual WSN nodes, which is typically not provided by the aforementioned models. With the exception of SINA, none of the models allow the programmer to express instructions that should be executed on individual nodes (e.g., to steer an actuator). A WSAN programmer has no other option than relying on lower-level and more complex models, which allow the behavior of the individual nodes to be programmed. In the following section, we will discuss some of these models.

2.2 Getting More Control over Individual Nodes

Next to traditional, sense-only WSN applications, wireless sensor- and actuator network (WSAN) applications are an emerging trend. Due to technological advancements, sensor nodes become more powerful and become able to steer actuators to control their environment. Applications embracing these technological advancements are often called *sense-and-react* applications [1]. In these applications, sensor data processing is typically moved inside the network, which requires complex node interactions to be programmed.

For developing sense-and-react applications, the aforementioned programming models are not appropriate. A high degree of control over the individual

sensor nodes is required, which is most often abstracted away in these models. While developers of sense-and-react applications require control over the individual nodes to steer actuators, they also require support for programming the individual nodes' interactions, which are essential in the setting of sensor networks. Programming these interactions is one of the most difficult challenges of WSANs [2].

In this section, we will discuss some representative node-level programming models, solutions to support the specification of node interactions, and models that allow the sensor network to be programmed from a macroscopic viewpoint while still offering control over the individual nodes.

2.2.1 Node-centric Programming

Models for node-centric programming allow individual sensor nodes to be programmed. As such, the programmer is offered a high degree of control over a node's behavior. However, these programming models are quite low-level. For instance, interactions with other nodes have to be programmed using explicit and physical addressing. In addition, not all models support reliable radio connections. As two representative models, this section briefly discusses the programming models offered by TinyOS [20] and SunSPOT [21]. In the context of this thesis, we will focus on how interactions can be programmed.

2.2.1.1 TinyOS

A sensor node running the TinyOS operating system can be programmed using nesC [22] [1]. nesC is an event-driven programming language derived from C in which *interfaces* are one of the most central concepts. An interface lists function signatures that are either tagged as *commands* or *events*. While command signatures represent operations, event signatures represent the functions that will be invoked by operations to answer their results (i.e., event handlers). A nesC application consists of several *components* that interact by *providing* or *using* these interfaces. While a component that provides an interface implements its commands, a component that uses an interface implements its event handlers. Note that a nesC application's components are statically fixed at compile-time and cannot be changed at runtime.

To allow node interactions to be programmed, Active Messages [23] provides a set of interfaces containing basic nesC communication primitives. `AMSend` and `Receive` are the most important interfaces and are shown in Listing 2.5. Other interfaces allow the low-level configuration of the communication primitives (e.g., by setting the transmission power).

```
1 interface AMSend {
2   command uint8_t    maxPayloadLength()
3   command void*      getPayload(message_t* msg, uint8_t len)
4   command error_t    send(am_addr_t addr, message_t* msg, uint8_t len)
5   command error_t    cancel(message_t* msg)
6   event   void       sendDone(message_t* msg, error_t error)
7 }
8 interface Receive {
9   event   message_t* receive(message_t* msg, void* payload, uint8_t len)
10 }
```

Listing 2.5: nesC: Communication Interfaces

To illustrate the use of nesC and these interfaces, we can consider a component that broadcasts its temperature reading in its one-hop neighborhood and prints out the temperature readings it receives. The code for this component, modified from [1], is shown in Listing 2.6.

```
1 module SampleAndPrint {
2   uses interface Boot;
3   uses interface TemperatureSensor;
4   uses interface AMSend;
5   uses interface Receive;
6 }
7
8 implementation {
9   message_t packet;
10
11   event void Boot.booted() {
12     call TemperatureSensor.read();
13   }
14
15   event void TemperatureSensor.readDone(uint16_t temp) {
16     uint16_t* packet_payload = (uint16_t*)(call AMSend.getPayload(&packet, NULL));
17     *packet_payload = temp;
18     call AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(uint16_t));
19   }
20
21   event void AMSend.sendDone(message_t* msg, error_t error) {
22     if(msg == &packet && error = SUCCESS) {
23       printf("Successfully sent temperature");
24     }
25   }
26
27   event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
28     if(len == sizeof(uint16_t)) {
29       printf("Received temperature %d", *payload);
30     }
31     return msg;
32   }
33 }
```

Listing 2.6: nesC: Communication Example

To start its application logic, this component uses the `Boot` interface, which signals a `booted` event when the TinyOS system is successfully started. The event handler for this event is implemented on lines 11 to 13 and calls the asynchronous `read()` operation on the `TemperatureSensor` to obtain the current temperature. Note that `read()` is a *split-phase* operation, which means that it returns its result by invoking an event handler (i.e., `readDone`). In this event handler, the measured temperature is encapsulated in a packet and broadcasted using `AMSend`'s `send` operation (lines 15-19). When this operation finishes, the `sendDone` event handler will be invoked and a debug message is printed (lines 21-25). Note that other components on the sensor node can also use `AMSend`'s `send` operation, which would cause this `sendDone` event handler to be invoked, too. Therefore, it is verified whether the sent packet is the one originating from this component before printing the debug message. Finally, the `receive` event handler will be invoked whenever a packet is received from the network. In this event handler, the packet is parsed and its contents are printed if it contained a temperature (lines 27-32).

Depending on the sensor hardware, a specific Active Messages implemen-

tation has to be used. However, in general, the implementations only support unreliable one-hop unicast or broadcast transmissions.

2.2.1.2 SunSPOT

SunSPOT sensor nodes (SunSPOTs) can be programmed in Java ME. In contrast to nesC, this allows the object-oriented programming paradigm to be used for programming WSN applications. To write node-level sensor code, it suffices to create a subclass of the `MIDlet` base-class. A *MIDlet* can be compared to a component from nesC and one or more MIDlets can run next to each other on a single SunSPOT. However, as every MIDlet runs in isolation (i.e., in an *isolate*), MIDlets do *not* share mutable memory. Note that, unlike nesC's components, MIDlets can be dynamically deployed, started, paused, stopped, and undeployed (i.e. at runtime).

For programming node interactions, either a reliable streaming or an unreliable datagram-style connection can be employed. These connections allow a node to interact with other nodes in its one-hop neighborhood. As in nesC, a node's transmission power can also be adjusted. Listing 2.7 illustrates the use of the datagram-style connection (i.e., a *radiogram* connection) by showing the code for a MIDlet that broadcasts its temperature reading in its one-hop neighborhood and prints out the temperature readings it receives.

```
1 public class SampleAndPrint extends MIDlet {
2   protected void startApp() throws MIDletStateChangeException {
3     try {
4       int temp = EDemoBoard.getInstance().getADCTemperature().getValue();
5
6       DatagramConnection dgConnection = null;
7       Datagram dg = null;
8
9       dgConnection = (DatagramConnection)Connector.open("radiogram://broadcast:50");
10      dg = dgConnection.newDatagram(dgConnection.getMaximumLength());
11      dg.writeInt(temp);
12      dgConnection.send(dg);
13      System.out.println("Successfully sent temperature");
14
15      dgConnection = (RadiogramConnection) Connector.open("radiogram://:50");
16      dg = dgConnection.newDatagram(dgConnection.getMaximumLength());
17      while(true) {
18        dg.reset();
19        dgConnection.receive(dg);
20        temp = dg.readInt();
21        System.out.println("Received temperature " + temp);
22      }
23    } catch(IOException ioe) {
24      /* Error */
25    }
26  }
27
28  protected void pauseApp() { }
29  protected void destroyApp(boolean b) throws MIDletStateChangeException { }
30 }
```

Listing 2.7: SunSPOT: Communication Example

As soon as the MIDlet is started, its `startApp()` method will be invoked. Remember that a MIDlet can also be paused or destroyed, upon which the

`pauseApp()` and `destroyApp()` methods will be invoked, respectively. However, as these methods are irrelevant for this MIDlet, their implementations are left blank. When the MIDlet starts, it obtains a temperature reading by reading out a temperature sensor (line 4). Afterwards, a datagram connection is initialized for broadcasting and a datagram encapsulating the measured temperature is sent over the node's radio (lines 6-13). Finally, to receive and print other temperature readings, a new datagram connection is initialized and listened upon (lines 15-22). For identifying the temperature datagrams, they are broadcasted and listened for on channel 50, as specified in the connection initialization statements (lines 9,15). Note that the `send` and `receive` methods are blocking and will only return when a new temperature datagram has been successfully sent or received. In case something goes wrong, these methods throw an exception.

2.2.1.3 Evaluation

Node-centric programming models offer a high degree of control over individual sensor nodes as they allow to explicitly specify these nodes' behavior. However, programming using these models is quite low-level. The programmer is responsible to write energy-efficient code and complex interactions among nodes might require the implementation of advanced routing algorithms on top of the very basic, typically one-hop, communication primitives. Even when interactions are limited to a node's one-hop neighborhood, they can become very complex as several messages might have to be exchanged with the neighboring nodes (e.g., for in-network processing). In these cases, application logic (e.g., controlling an actuator) and interaction logic (e.g., encapsulating messages in packets and parsing packets upon reception) become tangled, rendering the code difficult to understand, maintain, extend or reuse. To deal with these problem, several solutions, typically to be used in combination with these node-centric models, have been proposed. The following section discusses some of the most representative ones.

2.2.2 Support for Interactions among Nodes

Programming interactions among nodes, essential in the setting of sensor networks, is one of the most difficult challenges of WSANs [2]. As discussed in the previous section, basic communication primitives are often not appropriate to model rich interactions. To this end, several solutions have been proposed to facilitate programming these interactions. This section discusses some of the most representative ones, which either support node communication through message exchanges or date sharing.

2.2.2.1 Communication Through Messages

Typically, sensor nodes communicate by exchanging messages. At a low level, messages are represented as packets that are physically addressed to other nodes in a node's one-hop neighborhood. To facilitate communication through messages, solutions like Logical Neighborhoods [24] allow messages to be logically addressed to groups of nodes anywhere in the network. Solutions like LooCI [25], on the other hand, abstract messages as events that can be ex-

changed throughout the network by using a Publish-Subscribe interaction pattern [13].

Logical Neighborhoods. LOGICAL NEIGHBORHOODS is a programming abstraction that allows a node to define its neighborhoods based on logical properties of the nodes in the network. This extends the previously mentioned notion of neighborhood, which was based on a node's radio communication range. Depending on the definition of a neighborhood, some nodes will be in it, while other nodes won't. Note that these nodes might be located anywhere in the network, and not necessarily in the one-hop neighborhood of the node defining the neighborhood. Neighborhoods can be defined using a declarative programming language, called SPIDEY, which is an extension to an existing node-centric programming language (e.g., SPIDEY has been built on top of TINYOS's NESC). After defining a neighborhood, variants of the underlying language's communication primitives (i.e., `send` in NESC) can be employed to send messages to its members.

As an example, we can consider a WSAN application for detecting and fighting fire. In every room, several smoke sensors have to be deployed, next to a single actuator node for controlling the sprinklers. To allow the actuator node to easily communicate with the smoke sensors in its room, it can define the `mySensors` neighborhood as shown in Listing 2.8.

```
1 neighborhood template SmokeSensors(loc)
2   with Function = "sensor" and
3        Type = "smoke" and
4        Location = loc
5
6 create neighborhood mySensors
7   from SmokeSensors(loc: "auditorium A")
8   max hops 3 credits 30
```

Listing 2.8: Logical Neighborhoods: neighborhood template and instantiation

A neighborhood definition consists of a *template* and an *instantiation*. In this example, the template `SmokeSensors(loc)` includes all smoke sensors located in `loc` (lines 1-4). As the actuator node is in auditorium A, it creates a `mySensors` neighborhood by instantiating the template with the appropriate location (lines 6-8). Note that this instantiation also specifies the maximum number of hops that a member of the neighborhood can be separated from the actuator, and a limit on the cost for communication in terms of *credits* (i.e., an application-defined notion of communication costs).

To allow attributes such as Function and Type to be used in a neighborhood definition, potential members of the neighborhood should export them. To this end, all potential members have to instantiate a *node-template*. Listing 2.9 shows the SPIDEY code to be used by a smoke sensor from the example.

```
1 node template Device
2   static  Function
3   static  Type
4   static  Location
5   dynamic BatteryPower
6
7 create node smoke1 from Device
8   Function as "sensor"
9   Type as "smoke"
```

```
10   Location as "auditorium A"
11   BatteryPower as getBatteryPower()
```

Listing 2.9: Logical Neighborhoods: node template and instantiation

While the node template (lines 1-5) lists all exported attributes and their nature (i.e., static or dynamic), its instantiation (lines 7-11) provides a node's values for these exported attributes.

LooCI. LooCI is an event-based WSN middleware that introduces both a *component infrastructure* and a *decentralized event-bus* to sensor networks. It allows multiple components to be deployed on the same sensor node and it allows these components to communicate by exchanging events through the decentralized event-bus. Events abstract messages; they have a particular type and a payload that contains their values. Components can both publish events and subscribe to certain event types. Based on the event-subscriptions (i.e., so-called *wirings*), the events published in the sensor network will be routed from the publishers to the subscribers. LooCI is available on the SunSPOT platform where its components can be compared to MIDlets. A component can publish events by using the `publish(Event)` method, and it can process the events it received by implementing an event handler. In addition, LooCI comes with a *gateway* application to run on a back-end entity, which allows to remotely deploy components on the nodes in the network, and to configure the wirings amongst these components. As a result, sensor networks using the LooCI middleware can be dynamically (re)configured.

As an example, Listing 2.10 shows the code for a component that acts as a temperature filter (taken from the LooCI website[1]). This component subscribes to temperature reading events, which it will re-publish only when they exceed a certain threshold. This component has to be *wired* to a component that publishes temperature readings and to other components that subscribe to these readings.

```
1  public class TempFilter extends LooCIComponent {
2    private final int TEMP_THRESHOLD = 20;
3
4    public TempFilter() {
5      super("TempFilter",                              // component type
6            new byte[]{EventTypes.TEMP_READING},      // publications
7            new byte[]{EventTypes.TEMP_READING});     // subscriptions
8    }
9
10   protected void start() { }
11   protected void stop() { }
12
13   public void receive(Event event) {
14     if (event.getType() == EventTypes.TEMP_READING) {
15       PayloadBuilder payload = new PayloadBuilder(event.getPayload());
16       int reading = payload.getIntegerAt(0);
17       System.out.println("[TempFilter] Received temperature: " + reading);
18       if (reading > TEMP_THRESHOLD) {
19         payload = new PayloadBuilder();
20         payload.addInteger(reading);
21         publish(new Event(EventTypes.TEMP_READING, getComponentID(),
22                           payload.getPayload()));
23       }
```

[1] http://code.google.com/p/looci/wiki/FirstComponent

```
24      }
25    }
26 }
```

Listing 2.10: LooCI: A Temperature Filter Component

As soon as the component is started or stopped, its `start` or `stop` method will be invoked, respectively. However, as no special actions are required on these moments, the implementation for these methods is left blank (lines 10-11). The component's main functionality is implemented in its event handler (i.e., the `receive(Event)` method), which will be invoked whenever a new event is received. In case the received event is a temperature reading, the measured temperature is extracted from the event's payload, compared to the temperature threshold, and published in a new temperature reading event if it exceeded the threshold (lines 13-25). Finally, the `super` call in the component's constructor specifies the component's name together with the events it publishes and the events it subscribes to (lines 4-8). Note, however, that this data in the constructor is only meta-data that can be read using the gateway. To achieve interactions between components, wirings have to be explicitly configured. Even though these wirings can be hard-coded within a component's implementation, they are typically configured from the gateway.

To illustrate how the above component can be used, Listing 2.11 shows a transcript of an interactive session with the gateway.

```
1 deploy TempSensor.jar 0000.0000.0000.0001
2 => 3
3 deploy TempFilter.jar 0000.0000.0000.0001
4 => 4
5 wireLocal 101 3 101 4 0000.0000.0000.0001
6 => true
7 wireToAll 101 4 0000.0000.0000.0001
8 => true
9 deploy TempDisplay.jar 0000.0000.0000.0002
10 => 3
11 wireFrom 101 4 0000.0000.0000.0001 101 3 0000.0000.0000.0002
12 => true
```

Listing 2.11: LooCI: Interacting with the gateway

This transcript assumes the existence of three components: a *temperature sensor*, a *temperature filter* and a *component for displaying temperatures*. For their interactions, these components all employ temperature reading events, which have event type 101. The components are deployed to two different nodes in the network (lines 1,3,9). Note that the gateway always answers a component's node-local identifier after deployment (lines 2,4,10). These identifiers have to be used for wiring the components. As both the temperature sensor and -filter have been deployed on the same node, they are wired using `wireLocal`. The command on line 5 states that every temperature event published by the temperature sensor has to be delivered to the temperature filter. The temperature filter's published events, on the other hand, should be delivered to all components wired to it, which is expressed using `wireToAll` (line 7). Finally, the component for displaying temperature events should obtain these events from the temperature filter and is therefore wired to it using `wireFrom` (line 11). Next to the illustrated commands, `wireTo` and `wireFromAll` can also be used for wirings components. While the first allows a component to be wired to

send its events to a specifically addressed other component, the latter allows a component to be wired to receive a certain event from any component.

2.2.2.2 Communication Through Data Sharing

Next to communicating by exchanging messages, sensor nodes can also communicate by sharing data. Solutions that support this kind of communication introduce a shared memory space for a particular region of the network in which nodes can read or write data. This shared data is commonly represented as shared variables or tuples. In this section, we will discuss three representative solutions: Abstract Regions [26], which allows variables to be shared and aggregated, TeenyLIME [27], which allows tuples to be shared and reacted upon through event handlers, and FACTS [28], which allows tuples to be shared and reacted upon through declarative rules.

Abstract Regions. Abstract Regions is a collection of communication primitives that provide data sharing and aggregation in a particular region of a sensor network. To share data within a region, a node can export named variables, which can be read remotely by other nodes within the region. In addition, when several nodes export variables with identical names, their values can be aggregated using an associative operator (e.g., sum, max or min). A node can define an arbitrary number of regions in terms of radio connectivity or geographic location (e.g., a region with all nodes within n radio hops, or the k nearest nodes). To support QoS requirements, the communication primitives also return some *quality measures* (e.g., the amount of nodes that participated in an aggregation). Based on these measures, low-level communication parameters can be tweaked through a *tuning interface*. Abstract Regions was implemented in nesC and employs a lightweight, thread-like concurrency model to provide blocking operations.

To illustrate the use of Abstract Regions, we can consider an object-tracking application. This application uses several sensor nodes to take periodic magnetometer readings. Every node shares its readings and its location, and individually verifies whether its reading exceeds a certain threshold. In this case, an object is detected and its centroid can be computed using the shared variables to estimate its location. Listing 2.12 shows the code to be deployed on the sensor nodes (modified from [26]).

```
1 location = get_location();
2
3 // Create a region with the 8 nearest neighbors
4 region = k_nearest_region.create(8);
5
6 while (true) {
7   reading = get_sensor_reading();
8
9   // Store data in shared variables
10  region.putvar(reading_key, reading);
11  region.putvar(reg_x_key, reading * location.x);
12  region.putvar(reg_y_key, reading * location.y);
13
14  if (reading > threshold) {
15    // Obtain the ID of the node that measured the highest reading (i.e., the leader)
16    max_id = region.reduce(OP_MAXID, reading_key);
17
```

```
18      // If I am the leader...
19      if (max_id == my_id) {
20        // Compute the centroid
21        sum = region.reduce(OP_SUM, reading_key);
22        sum_x = region.reduce(OP_SUM, reg_x_key);
23        sum_y = region.reduce(OP_SUM, reg_y_key);
24        centroid.x = sum_x / sum;
25        centroid.y = sum_y / sum;
26        send_to_basestation(centroid);
27      }
28    }
29    sleep(periodic_delay);
30  }
```

Listing 2.12: Abstract Regions: Implementing an object-tracking application

Every sensor node initializes a region to contain its eight nearest neighbors using the `k_nearest_region.create()` primitive (line 4). In the main loop (lines 6-30), the magnetometer is read out and the required data is stored in shared variables using the `putvar()` primitive (lines 7-12). If the reading happens to be above the predefined threshold, an object is detected and its position should be computed. However, to make sure that only a single node computes this position, a leader is elected based on the measured magnetometer readings. To this end, the `reduce()` primitive is employed to aggregate the measured readings using the `OP_MAXID` operator (line 16). Finally, the node verifies whether it was elected as the leader, and if so, it computes the centroid and sends it to the base station (lines 18-27).

TeenyLIME. TeenyLIME brings the *tuple space* abstraction, originally proposed in Linda [29], to WSNs. A tuple space (TS) is a repository of *tuples*, which represent data as a sequence of typed fields (e.g., $< "temp", 30 >$). Every node has a local TS, which is shared with other nodes in its one-hop neighborhood, and several operations can be performed on either the local TS, or the union of all TS's in the neighborhood (i.e., the shared TS). As in Linda, tuples can be stored using the `out` operation, read using the `rd` operation, and removed using the `in` operation. For the latter operations, tuples can be matched through *pattern matching*. A pattern is a tuple with fields for which the exact values have not been specified (e.g., $< "temp", ?int >$). In contrast to Linda, fields in a pattern can also be constrained by a predicate and a pattern can be constrained only to match tuples of a certain age. In addition, tuples can be set to expire, after which they are automatically removed from the TS. Next to the above operations, *reactions* can also be registered on a TS, and the notion of *capability tuples* allows a node to store a pattern in its TS, which it can then instantiate whenever it is matched by a query from another node. The latter can be used, for instance, to make a node read out its sensors only on demand. TeenyLIME has been implemented on top of TinyOS and provides an API for nesC. As all operations are *split-phase* operations, they return their results by invoking an event handler.

To illustrate the API, we can consider an application for detecting fire, which bases its detection on very high temperature readings. Under the assumption that all temperature sensors are in the one-hop neighborhood of an actuator that controls sprinklers, the code from Listings 2.13 and 2.14, modified from [27], can be deployed on the sensor- and actuator nodes, respectively.

```
1 event void Boot.booted() {
2   call SensingTimer.start(TIMER_REPEAT, SENSING_TIMER);
3 }
4
5 event result_t SensingTimer.fired() {
6   return call TemperatureSensor.read();
7 }
8
9 event void TemperatureSensor.readDone(uint16_t temp) {
10  tuple tempTuple = newTuple(2, actualField_uint16(TEMPERATURE),
11                                actualField_uint16(temp));
12  setExpireIn(tempTuple, 3);
13  call TupleSpace.out(FALSE,TL_LOCAL,&tempTuple);
14  return SUCCESS;
15 }
```

Listing 2.13: TeenyLIME: code for a sensor

```
1 TLOpId_t tempTupleReaction;
2
3 event void Boot.booted() {
4   tuple tempPattern = newTuple(2, actualField_uint16(TEMPERATURE),
5                                   greaterField(TEMPERATURE_SAFETY));
6   tempTupleReaction = call TS.addReaction(TRUE, TL_NEIGHBORHOOD,
7                                           &tempPattern);
8 }
9 event result_t TS.tupleReady(TLOpId_t opId, tuple* tuples, uint8_t number) {
10  if(opId == tempTupleReaction) {
11    // Turn on sprinklers...
12  }
13 }
```

Listing 2.14: TeenyLIME: code for an actuator

As soon the temperature sensor is started, it sets a timer to be fired periodically (Listing 2.13, lines 1-3). When this timer fires, the current temperature is measured, encapsulated in a tuple and stored in the local tuple space, as specified by `TL_LOCAL` in the `out` operation (lines 5-15). Note that the tuple is set to expire after 3 *epochs*[2] (line 12). The actuator node, on the other hand, configures its TS when it is started. To this end, a pattern for matching temperature tuples with a value exceeding the predefined threshold is created (Listing 2.14, lines 4-5) and used to register a reaction on the shared TS, as specified by `TL_NEIGHBORHOOD` in the `addReaction` operation (lines 6-7). As soon as a tuple in the shared TS matches this pattern, the `tupleReady` event will be fired, and the sprinklers can be turned on (lines 9-13). Note that both `out` and `addReaction` take a boolean as a first argument (Listing 2.13, line 13, Listing 2.14, line 6). This boolean indicates whether the tuples for the operation should be communicated reliably. As it is important for the actuator's reaction to be triggered when a high temperature is observed, this boolean is set to true for the reaction. In addition, note that every operation in the TeenyLIME API returns an *operation id*, which can be used in the event handler to identify the operation that completed (Listing 2.14 lines 6,10). Finally, even though tuples can expire, it is not possible to react to their expiration and therefore special application logic would be required to decide when to turn off the sprinklers from the example.

[2] An epoch is a constant amount of time specified by TeenyLIME.

FACTS. FACTS introduces a rule-based programming language to WSNs. Its main abstractions are *facts*, *rules*, and *functions*. Facts are named tuples consisting of key-value pairs and are stored in a node-local *fact repository*. In contrast to TeenyLIME's tuple space, the fact repository itself is not shared, but a fact can be sent to other nodes (possibly multiple hops away) which will then add it to their fact repository. As such, a fact repository can contain facts originating from several nodes. Whenever a fact is added to the fact repository, it can trigger the execution of rules, which specify a reaction to facts. Rules are named and consist of a set of *conditions* and an ordered list of *statements* to be executed whenever their conditions are met. While a condition can verify a boolean expression or the presence of a particular fact in the fact repository, a statement can call a C function or create, remove, modify or publish facts. For instance, a C function can read out a sensor and the result can be stored in a new fact. Both conditions and statements can take a *slot* as a parameter, which has to be used to select relevant facts from the fact repository. As such, conditions and statements are entirely independent: while a parametrized condition is satisfied when at least one fact matches its slot, a parametrized statement is executed for every fact matching its slot. Finally, FACTS encapsulates related rules and facts in a *ruleset*, which can be compared to a component that provides a certain service. Rulesets can use other rulesets, which fosters code reuse.

As an example, we can consider the implementation of a coverage algorithm. In this algorithm, every node broadcasts its range and individually verifies whether it covers the range of any other node. Based on this knowledge, some sensor nodes might, for instance, be put to sleep. Listing 2.15 shows the implementation for this algorithm (modified from [28]), which can be deployed on every sensor node in the network. A rule is written as its name, followed by its conditions prefixed with `<-` and its statements prefixed with `->`.

```
1 sendRange
2 <- Exists Timer.expiredSlot
3 -> Retract Timer.expiredSlot
4 -> Define "rangeFact"
5 -> Set ("rangeFact" "xMin") (posXSlot - System.txRadiusSlot)
6 -> Set ("rangeFact" "xMax") (posXSlot + System.txRadiusSlot)
7 -> Set ("rangeFact" "yMin") (posYSlot - System.txRadiusSlot)
8 -> Set ("rangeFact" "yMax") (posYSlot + System.txRadiusSlot)
9 -> Send 0 System.txPowerSlot
10        ("rangeFact" [(("rangeFact" "owner") == nodeIDSlot)])
11 -> Define "rangeSendFact"
12
13 xyMinCovered
14 <- Exists "rangeSendFact"
15 <- Eval ((posXSlot - System.txRadiusSlot) < ("rangeFact" "xMin"))
16 <- Eval ((posYSlot - System.txRadiusSlot) < ("rangeFact" "yMin"))
17 -> Define "xyMinCoveredFact"
18
19 determineCoverage
20 <- Exists "xyMinCoveredFact"
21 <- Exists "xMaxYMinCoveredFact"
22 <- Exists "xyMaxCoveredFact"
23 <- Exists "xMinYMmaxCoveredFact"
24 -> Define "coveredFact"
```

Listing 2.15: FACTS: a coverage algorithm

The `sendRange` rule is triggered by a timer that adds a fact to the fact repository upon firing (lines 1-2). As a reaction, the rule removes the timer's fact

and creates a new `rangeFact` to contain the node's range (lines 3-8). This fact is broadcasted to all nodes in the one-hop neighborhood (line 9-10). Note that only the locally created fact has to be broadcasted to avoid re-broadcasting `rangeFact`s from other nodes. To this end, the fact to be broadcasted is selected from the fact repository using a slot with an additional condition on the fact's owner, expressed between brackets (line 10). After broadcasting the fact, a `rangeSendFact` is added to the fact repository to indicate that the range information was published (line 11). Afterwards, a node awaits the `rangeFact`s from its neighbors. Whenever such a fact is received, the `xyMinCovered`, `xMaxYMinCovered`, `xMinYMaxCovered`, and `xyMaxCovered` rules will inspect its data to verify whether the sensor node is covering the received range. For brevity, only the `xyMinCovered` rule is shown (lines 13-17). Note that this rule adds a `xyMinCoveredFact` to the fact repository if the bottom-left of the received range is covered (line 17). When all corners of the received range are covered, the range is covered and a `coveredFact` is added to the fact repository (lines 19-24). This fact can be used in other rules to act accordingly.

2.2.2.3 Evaluation

This section introduced several representative solutions that aid in specifying interactions among sensor nodes when programming sensor network applications from a node-level perspective. To this end, it categorized the solutions according to the type of communication that they support.

While the solutions for *communication through messages* clearly facilitate sending messages, they do not help in processing these messages upon reception. Indeed, both Active Messages and LooCI require all received messages (or events) to be parsed and processed in a single event handler. Clearly, this approach won't scale as a node participates in more interactions (e.g., for in-network processing of various data from several nodes).

For the solutions supporting *communication through data sharing*, more variations were presented. For instance, Abstract Regions allows to elegantly express algorithms that operate on data originating from several nodes in the network through aggregation. As a result, it is very appropriate to program in-network processing logic. However, as it does not provide any means to react to the appearance of new data, there's no other option than to actively pull the shared data in order to process it. TeenyLIME, on the other hand, allows to react to the appearance of new data in the shared tuple space. Nevertheless, it suffers from the exact same issues as were identified for the solutions for communication through messages: all registered reactions have to be dealt with in a single event handler. Despite this fact, TeenyLIME also introduced some interesting ideas. For instance, as tuples commonly contain sensor data, typically only useful for a limited amount of time, the introduction of tuple expiration and the ability to react to tuples with a maximum age is clearly well-founded. The ability to react to the expiration of tuples can be considered as a missing feature. Finally, FACTS' rules allow to modularize interactions. When a node has to react differently to various facts (i.e., when it participates in multiple interactions), it suffices to install multiple rules. In addition, unlike TeenyLIME, FACTS allows to react to the appearance of *multiple* facts (e.g., a temperature- *and* a humidity fact). However, variables can be considered as a missing feature. In the absence of variables, no facts can be related by their

content. For instance, it's not possible to express a rule to compute a comfort level based on a temperature sensor reading and a humidity sensor reading that originate from the same node or room. Terfloth et al. motivate the absence of variables by the fact that binding these variables at runtime would be too expensive in terms of memory usage for an embedded system [28]. Noteworthy, however, is the idea to modularize rules in rulesets that can be reused by other rulesets.

2.2.3 Network-centric Programming

Having introduced node-centric programming models and solutions that support programming node interactions from a node-level perspective, we shift our focus to the programming models that allow a sensor network to be programmed from a macroscopic viewpoint while still offering control over the individual nodes. These programming models are often referred to as *network-centric-* or *macroprogramming* models [2]. As they allow the desired global behavior of the entire sensor network to be programmed in a centralized fashion, the programmer can maintain a high-level overview of the application.

Network-centric programming models can be categorized according to their target applications, which can be either *applications exhibiting logical homogeneity*, *applications requiring mobile code* or *applications exhibiting logical heterogeneity*. This section briefly introduces one representative model for each category and thereby completes the picture of the major approaches in the state of the art for programming wireless sensor networks.

2.2.3.1 Models for Homogeneous Applications

In applications that exhibit logical homogeneity, every sensor node requires the same behavior. Typical examples are applications for constructing a routing tree or for tracking objects. Among the programming models that support the development of this kind of applications are Kairos [30] and snLog [31]. While snLog is a declarative, rule-based language, Kairos is a set of abstractions that can extend any imperative language. As a representative example for this category of programming models, we will briefly discuss the latter.

Kairos abstracts a wireless sensor network as a collection of nodes that can all be programmed together in a single program. To this end, it extends any node-centric programming language with three abstractions: an abstraction for a *node*, typically a member of a *node-set* that can be iterated over, an abstraction for *a list containing a node's one-hop neighbors* (i.e., a node-set), and an abstraction for *remote (read-only) access to a node's variables*. Using these abstractions, a WSN application can be programmed in a way that resembles the high-level algorithm descriptions found in textbooks. Kairos comes with a *precompiler* to compile its programs to code for the extended node-level language, and a *runtime* library for this language. Both the precompiler and the runtime have been implemented as an extension to Python. After precompiling a Kairos program and compiling the node-level code, the binary can be deployed on every node in the network. By default, the Kairos runtime *loosely synchronizes* state across nodes, which means that reading a remote variable initially blocks until the value is obtained and cached, but afterwards immediately re-

turns the cached value. To reduce communication overhead, these cached values are lazily updated as the remote variables are modified, and as such, eventually converge to the most recent values.

To illustrate Kairos' programming model, we can consider an application for constructing a shortest-path routing tree. In this program, every node periodically queries its neighbors for their distance to the root, selects the node at the shortest distance, and updates its own parent node accordingly. Listing 2.16 shows the code for this application (modified from [30]).

```
1 void buildtree(node root)
2   node parent, self;
3   unsigned short dist_from_root;
4   node_list neighboring_nodes, full_node_set;
5   unsigned int sleep_interval=1000;
6   // Initialize every node
7   full_node_set=get_available_nodes();
8   for (node temp=get_first(full_node_set);
9        temp!=NULL;
10       temp=get_next(full_node_set))
11    self=get_local_node_id();
12    if (temp==root)
13      dist_from_root=0;
14      parent=self;
15    else
16      dist_from_root=INF;
17    neighboring_nodes=create_node_list(get_neighbors(temp));
18  // Start the main loop on every node
19  full_node_set=get_available_nodes();
20  for (node iter1=get_first(full_node_set);
21       iter1!=NULL;
22       iter1=get_next(full_node_set))
23    for(;;)
24      sleep(sleep_interval);
25      for (node iter2=get_first(neighboring_nodes);
26           iter2!=NULL;
27           iter2=get_next(neighboring_nodes))
28        if (dist_from_root@iter2+1<dist_from_root)
29          dist_from_root=dist_from_root@iter2+1;
30          parent=iter2;
```

Listing 2.16: Kairos: Constructing a shortest-path routing tree

Lines 2 to 5 declare the variables relevant to the application. These variables can be used in the context of the global program (e.g., `full_node_set`), or in the context of a node (i.e., within an iteration over all nodes). By iterating over all nodes in the sensor network, obtained from `get_available_nodes()` (line 7), every node's state is first initialized (lines 8-17). Only the root node knows its distance to the root, and its parent (i.e., itself). In addition, every node's neighboring nodes are obtained using `get_neighbors(Node)` (line 17). Afterwards, the main loop is started on every node (lines 18-30). After a period of sleep, the distance from each neighbor to the root is remotely read using `dist_from_root@iter2` (line 28), and a neighbor is set as a node's parent if it provides a shorter path to the root than the currently stored one (lines 28-30). Note that, due to the loose state synchronization, the remotely read distance is not necessary the most recent one. However, as every node keeps looping to update its state, every node will eventually obtain the right parent on the shortest path to the root node.

2.2.3.2 Models for Mobile Code

Certain applications or network tasks can be elegantly expressed as *agents* that move throughout a WSN. The implementation of such an agent is often called *mobile code* [1]. After injecting an agent in a WSN, it autonomously moves throughout the network to execute instructions on every visited node, and afterwards typically returns to the node where it was injected to return some values. Among the programming models that support the development of mobile code are ActorNet [4], Spatial Programming [32] and SpatialViews [33]. ActorNet allows mobile code to be written in a Scheme-like language, and it allows this code to migrate from a node to any other node in its one-hop neighborhood. Spatial Programming and SpatialViews, on the other hand, allow regions in the WSN to be specified and allow mobile code to migrate to the nodes within these regions. Note that, as the position of the nodes might change, their membership in a particular region can also change. Since both Spatial Programming and SpatialViews expect the nodes to have substantial computation and communication resources, an unrealistic scenario in WSNs, we will discuss ActorNet as a representative example for this category of programming models.

ActorNet is a Scheme-like language that allows programming mobile agents. To this end, it implements the actor model of distributed computing [34] on WSNs. An agent is represented by an *actor*, which can migrate itself to another node in its one-hop neighborhood. In addition, every node runs, by default, a single *launcher* actor. A launcher actor can be sent expressions which it will evaluate. As such, an actor can evaluate expressions on the nodes in its one-hop neighborhood, which can communicate their result back to their originating actor through message passing. In ActorNet, actor migration is implemented in terms of the `callcc` primitive, which takes a single-parameter function as its operand, and calls this function with the *current continuation*. Because the state of an actor is represented by its current continuation and the value that will be passed to it, an actor can migrate to a neighboring node by sending its continuation and its value to that node's launcher actor. As a result, the launcher actor will evaluate the continuation and continue the migrated actor's work. The implementation for the `migrate` primitive, modified from [4], is shown below:

```
1 (define (migrate addr value) ; Migrate to actor addressed by addr
2   (callcc (lambda (cc)
3     (send addr cc value))))
```

The `send` primitive is used to send a message to the launcher actor with address *addr* (line 3). Note that `send` can also be used to send a message to a regular actor, which can access its messages in a queue accessible through the `msgq` primitive.

To illustrate ActorNet's programming model, we can consider an actor that searches a WSN for the node where the highest temperature is measured, turns on a LED on this node, and returns the measured temperature to the base station to print it. The code for this *searching actor*, modified from [4], is shown in Listing 2.17.

```
1 (rec (measure (oriActor)          ; Return temp and launcher actor id
2   (send oriActor (io 1) (io 0)))) ; to originating actor
```

```
3
4 (rec (move returnPath maxTemp)    ; Searching Actor
5   (seq
6     ;;evaluate measure on the neighbors
7     (send 0 measure (id))
8     (delay 100) ;;wait for 10 sec
9     ((lambda (maxFromNeighbors)
10       (par
11         ;;if it arrives at an maximal point
12         (cond (<= (car maxFromNeighbors) maxTemp)
13         ;;then turn on LED and return the maxTemp to the base station
14           (seq (io 2)
15                (return migrate returnPath maxTemp))
16         ;;else migrate to the highest temp. neighbor
17           (move (cons (io 0) returnPath)
18                 (migrate
19                    (cadr maxFromNeighbors)   ; neighbor's launcher ID
20                    (car maxFromNeighbors)))) ; neighbor's temp
21         (setcdr (msgq) nil))) ; reset msgq
22     ; find the max temp neighbor
23     (max (cdr (msgq)) (list 0 0)))))
24
25 (rec (return migrate path temp)
26   (cond (equal path nil)
27     (print temp)
28     (return migrate (cdr path) (migrate (car path) temp))))
```

Listing 2.17: ActorNet: An agent/actor looking for the highest temperature in the WSN

The searching actor is implemented by the `move` function (lines 4-23). To obtain the temperatures in its surroundings, it evaluates `measure` on its neighboring nodes and waits ten seconds to process the results (lines 6-8). The `measure` function sends the measured temperature, obtained through `(io 1)`, and the id of the node's launcher actor, obtained through `(io 0)` back to the searching actor, whose address was passed to the measure function using `(id)` (line 7). After ten seconds, the searching actor iterates over its message queue to obtain the message from the neighboring node that answered the highest temperature (lines 22-23). It then verifies whether this temperature is higher than its currently found maximum temperature (lines 11-20). If it is, the searching actor migrates to this neighbor to continue its search for even higher temperatures (lines 17-20). On the other hand, when none of its neighboring nodes answered a higher temperature than the current maximum, the searching actor is on the node with the highest temperature, turns on the LED using `(io 2)`, and returns the found maximum temperature to the base station (lines 14-15). To this end, it employs the `return` function (lines 25-28) to migrate back to the base station by following the path that was stored on its search (line 17) back, until the base station is reached and the temperature can be printed (line 27). Note that the `par` primitive is used on line 10 to reset the message queue (line 21) in parallel with the processing of the message from the highest neighbor.

2.2.3.3 Models for Heterogeneous Applications

In applications that exhibit logical heterogeneity, various nodes need to behave differently. A typical example is an application where sensor nodes need to collect sensor data and actuator nodes need to process this data in order to

control their actuators. One network-centric programming model that supports the development of these applications is ATaG [35, 36].

Abstract Task Graph (ATaG) is a framework providing a mixed declarative-imperative programming paradigm for developing WSN applications from a macroscopic viewpoint. The main abstractions offered are *abstract tasks*, *abstract data items* and *abstract channels*. Tasks represent an application's processing and can produce or consume data items. Channels, on the other hand, represent the interactions among tasks. They connect data items to the tasks that produce and consume them. While tasks, data items, and channels have to be declared declaratively (i.e., graphically in a task graph), tasks have to be implemented in an imperative language. Therein, the programmer can use two primitives: `getData()` to consume a data item, and `putData()` to produce a data item. Based on the channel declarations, ATaG's runtime will move these data items between producers and consumers. Finally, given the concrete information of the target network, the ATaG compiler can be used to transform an ATaG program into architecture-specific node-level behaviors and decide on their deployment to nodes.

To illustrate this programming model, we can consider a simple application in which temperature sensors periodically take readings to be processed by several cluster-heads, each deployed in a particular region. Figure 2.1, taken from [1], depicts the task graph for the corresponding ATaG program. In this graph, tasks are represented by ellipses, data items are represented by rectangles, and channels are represented by arrows. As expressed by the channels,

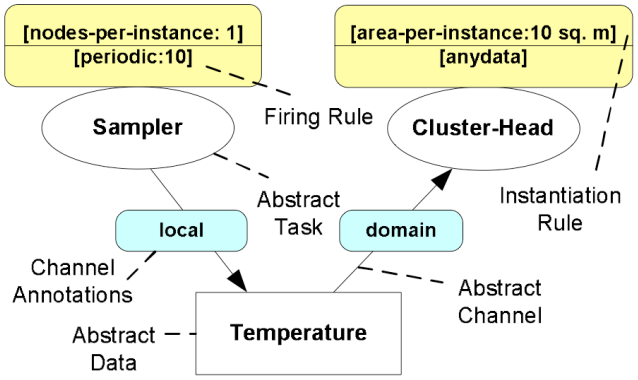


Figure 2.1: ATaG: Cluster-based data processing - task graph

the *Sampler* tasks produce *Temperature* data items, which are consumed by a *Cluster-Head* task. Note that both tasks and channels are annotated. These annotations are shown in colored squashed rectangles. A task's annotation has two rules: an *instantiation rule* deciding the task's instantiation to nodes, and a *firing rule* deciding when the task will be executed. For instance, the *Sampler* task is declared to be instantiated on every node in the network (i.e., through `nodes-per-instance:1`), while the *Cluster-Head* is declared to be instantiated once every 10 square meters (i.e., through `area-per-instance:10sq.m`). In addition, the *Sampler* task will be executed every 10 seconds, as specified by the `periodic` firing rule, while the *Cluster-Head* will be executed whenever any

of the data items it consumes becomes available, as specified by the `anydata` firing rule. If a task consumes multiple data items, the `all-data` firing rule can also be used to specify that the task should be executed as soon as all of these data items are available. Channels are annotated to express the specific interest of a task in a data item. For instance, the *Temperature* data items produced by a *Sampler* task remain `local` to the node where it was produced, while the *Cluster-Head*s will gather these data items from the *Sampler* tasks in their `domain`. Roughly speaking, a *Cluster-Head*'s *domain* consists of all nodes hosting a *Sampler* task within 10 square meters around the *Cluster-Head* (i.e., the domain is decided by the task's `area-per-instance` instantiation rule).

Based on this task graph, the ATaG compiler will generate templates for the tasks that can be implemented by the programmer. For instance, the code for the *Sampler* task, taken from [1], is shown in Listing 2.18. The template itself contained only the loop with the sleep instruction, based on the `periodic` firing rule. Similarly, the template for the *Cluster-Head* contains an event handler in which `getData()` can be used to consume the available data items. This event handler will be invoked when the *Temperature* data item consumed by the task becomes available.

```
1 // ...
2 while (TRUE) {
3   sleep (10000);
4   // Written by the programmer
5   int temperature = getTemperature();
6   TemperatureDataItem t = newTemperatureDataItem(temperature);
7   putData(t);
8 }
9 // ...
```

Listing 2.18: ATaG: Cluster-based data processing - code for the Sampler task

Note that ATaG programs are highly extensible and reusable because there is no direct task-to-task coupling. The ATaG framework allows users to build libraries of ATaG programs that can be reused to build larger applications.

2.2.3.4 Evaluation

Network-centric programming models allow WSN applications to be programmed in a centralized fashion, thereby allowing programmers to maintain a high-level overview of their application. In this section, we introduced several network-centric programming models categorized by their application domain. Given the fact that these solutions each provide the appropriate level of abstraction for their target applications, the growing interest in macroprogramming of sensor networks [35] is well-founded. Next to facilitating WSN application development, these models also foster code reuse. Even ATaG, the model that targets applications exhibiting logical heterogeneity, allows tasks and interactions to be reused among various projects. It does, however, still require the programmer to deal with the complexities of event handlers.

2.3 Overview

The table in Figure 2.2 gives an overview of the state of the art for programming WSNs, as presented in this chapter. While the first columns of this table present

the general properties of each solution, the remaining columns require additional explanation.

Communication. The column for communication presents the properties of the solutions' features that support node interactions. It is only filled in if these features are visible to the programmer, otherwise the properties are presented as non-applicable. The *addressing* property specifies how the nodes to interact with have to be addressed, which can be either *physical* (e.g., by their MAC address or distance in amount of hops) or *logical*. Through logical addressing, nodes are addressed by their application-level properties (e.g., all nodes collecting temperature measurements). Note that interactions have a certain *scope*. While some solutions allow a node to interact only with nodes in their one-hop neighborhood, others allow a node to interact with nodes anywhere in the network. The final communication property describes how a programmer has to process the data that was received from other nodes, for which several options are applicable:

- Single event handler: all received data has to be processed in a single event handler. Therein, the type of the received data has to be dispatched over to select the appropriate processing (i.e., received data has to be parsed).
- Blocking receive operation: a programmer has to anticipate the reception of a particular data type by invoking a blocking receive operation. Typically, multiple threads are required to receive various data types. The alternative consists of manually parsing the received data.
- Single message queue: all received data is accessible through a single message queue and has to be parsed.
- Event handler per data type: for each possible data type that can be received, an individual event handler has to implement its processing.
- Variable access: all received data is accessible through logical variables whose names specify the data type. Additionally, some solutions allow the values for a particular logical variable to be *aggregated* using a reduce operation.
- Declarative rules: rules can be used to specify the processing of one or more received data types.

Note that the parsing of received data causes logic concerned with various node interactions to be tangled, which is an issue in most programming models. As we will discuss in Chapter 3, our work (i.e., CrimeSPOT) addresses this issue by allowing the programmer to use declarative rules to specify the processing of received data.

Support for reuse. The final column of the table presents the means that are available to the programmer to reuse interaction- or application logic. While *interaction logic* consists of all code concerned with interactions with other nodes, *application logic* consists of all code concerned with the rest of the application (e.g., code for steering an actuator, or SQL-like code for extracting temperature

readings from the network). In both sub-columns, the *general* keyword represents the typical means for code reuse in the solution's programming paradigm (e.g., function declarations).

Note that only very few solutions introduce special means for reusing logic within or among applications. The only exceptions are FACTS, snLog and ATaG. As we will discuss in Chapter 4, our work (i.e., Macro-CrimeSPOT) can also be added to this list of exceptions.

2.4 Conclusion

This chapter gave a non-exhaustive overview of the state of the art for programming wireless sensor networks. Several representative programming models were introduced for several application domains.

For traditional, sense-only applications, we discussed programming models for *raw data extraction*, for *programming queries*, and for *interpreted data extraction*. These models introduce high-level abstractions that are well-suited for sense-only applications, yet too high-level for sense-and-react applications.

For sense-and-react applications (or so-called WSAN applications), a developer requires more control over the individual nodes to steer their actuators and to implement in-network processing logic. In the context of these applications, we discussed representative *node-centric- and network-centric programming models*.

Node-centric programming models allow individual WSN nodes to be programmed. As these models only provide limited support for programming interactions among nodes, several solutions can be used to complement them. In this category, we introduced solutions supporting *message communication* and *data sharing*. However, we had to conclude that these solutions typically don't help in processing data upon reception. One exception was FACTS, which allows the programmer to modularize node interactions using rules that can moreover be reused in several applications.

The discussed network-centric programming models allow WSNs to be programmed as a whole while still offering control over the individual nodes. In this category, we focussed on solutions supporting *homogenous applications*, *mobile code*, and *heterogenous applications*. We believe that these models introduce the appropriate level of abstraction for programming sense-and-react applications, while also fostering code reuse.

Solution	Primary Application Domain	Programming Paradigm	Node-level control	Communication			Processing received data	Support for reuse	
				Type	Transmitting Data			Int. Logic	App. Logic
					Addressing	Scope			
Raw Data Extraction									
TAG/COUGAR/TINYDB	sensing	Declarative: SQL-like	No	N/A	N/A	N/A	N/A	N/A	None
SINA	sensing	Declarative: SQL-like, Imperative	Yes	Message Passing	Physical	Network	Single event handler	None	None
Programming Query									
REGIMENT	sensing	Functional	No	N/A	N/A	N/A	N/A	N/A	General
Interpreted Data Extraction									
SEMANTIC STREAMS/DSWARE	sensing	Declarative: SQL-like	No	N/A	N/A	N/A	N/A	N/A	None
Node-centric Programming									
NESC(TINYOS)	sensing and reacting	Imperative	Yes	Message Passing	Physical	Neighbors	Single event handler	General	General
JAVA(SUNSPOT)	sensing and reacting	Imperative	Yes	Message Passing	Physical	Neighbors	Blocking receive operation	General	General
Support for Node Interactions									
LOGICAL NEIGHBORHOODS (NESC)	sensing and reacting	Declarative: Special-purpose	Yes	Message Passing	Logical	Network	Single event handler	General	General
LOOCI (JAVA)	sensing and reacting	Imperative	Yes	Message Passing	Physical	Network	Single event handler	General	General
ABSTRACT REGIONS (NESC)	sensing and reacting	Imperative	Yes	Data Sharing	Physical	Network	Variable access, Aggregation	General	General
TEENYLIME (NESC)	sensing and reacting	Imperative	Yes	Data Sharing	Physical	Neighbors	Single event handler	General	General
FACTS	sensing and reacting	Declarative: Rule-based	Yes	Data Sharing	Physical	Network	Declarative rules	Rule-sets	Rule-sets
CRIMESPOT (JAVA)	*sensing and reacting*	*Declarative, Imperative*	*Yes*	*Data Sharing*	*Physical*	*Network*	*Declarative Rules*	*General*	*General*
Network-centric Programming									
KAIROS	sensing and reacting	Imperative	Yes	Data Sharing	Physical	Neighbors	Variable access	General	General
SNLOG	sensing and reacting	Declarative: Rule-based	Yes	Data Sharing	Physical	Neighbors	Declarative rules	None	Built-in Predicates
SPATIALVIEWS/SPATIAL PROGRAMMING	sensing and reacting	Imperative, Mobile Code	Yes	N/A	N/A	N/A	N/A	N/A	General
ACTORNET	sensing and reacting	Functional, Mobile Code	Yes	Message Passing	Physical	Neighbors	Single message queue	General	General
ATaG	sensing and reacting	Declarative, Imperative	Yes	Data Sharing	Logical	Network	Event handler per data type	Task Graphs	General
MACRO-CRIMESPOT	*sensing and reacting*	*Declorative, Imperative*	*Yes*	*Data Sharing*	*Physical*	*Network*	*Declorative Rules*	*Macros, Imports*	*Imports*

Figure 2.2: An overview of the covered State of the Art

3

Node-centric Programming with CrimeSPOT

When programming wireless sensor network (WSN) applications from a node-level perspective using event-based WSN middleware, every sensor node has to be individually programmed by specifying its interactions with other sensor nodes. The nodes communicate exclusively via a decentralized event bus to which they publish events and from which they receive the published events of the types they subscribed to. Hence, sensor nodes' interactions often involve reacting to events published by other sensor nodes. However, the only means to process received events and specify reactions are *event handlers*, which have already been shown to violate several software engineering principles, including composability, scalability and separation of concerns [12].

In this chapter, we introduce CrimeSPOT, a new domain-specific programming language for programming individual WSN nodes. This language constitutes a building block, to be used on top of event-based middleware for WSNs, that allows the programmer to use *declarative interaction rules* to specify the interactions with other sensor nodes. This addresses the various problems that using an event handler for this purpose brings along. The programmer no longer has to dispatch a received event to its appropriate reaction, nor is she required to implement ad-hoc state management to keep track of which events have already been received and whether or not they are still valid.

After considering a representative running example and extracting programming support requirements, we will show how CrimeSPOT meets these requirements by discussing its architecture and all of its features. Simultaneously, we will incrementally implement the running example. To conclude this chapter, we will revisit the running example and finalize its CrimeSPOT implementation.

While, up till now, we discussed the problem and our solution in the light of plain event-based middleware, CrimeSPOT can also be used on top of event-based *component* middleware. Basically, a component encapsulates node-level application logic, while component middleware allows multiple components to

run next to each other on the same sensor node. From this point on, we will assume the use of event-based component middleware on top of a Java platform and talk about *components* rather than *sensor nodes* when discussing node-level application logic.

3.1 Running Example

To illustrate the need for programming support for building WSN applications using event-based component middleware, this section introduces a representative running example. In the next sections, we will introduce the CrimeSPOT language while referring to this running example to explain the language features.

As a representative event-based WSN application[1], we can consider a system that controls the heating and logs the comfort levels of festival tents. Such an application requires several components, each performing their own specific task. Sensor readings should be made available by temperature- and humidity sensor components, heaters should be controlled by heating controller components, and the application should be steered by a comfort level monitor component. Even though some of these components could be deployed on the same WSN node, for simplicity, we assume that every component is deployed on a different node. A high-level overview of the application is depicted in Figure3.1. Within every tent, three components have to be deployed: the *HeatingController*, the *TemperatureSensor* and the *HumiditySensor*. Typically, these components are deployed only once within a tent, but for larger tents, multiple redundant components could also be deployed. The *ComfortLevelMonitor* component, on the other hand, is deployed only once and controls the entire system. In the figure, a component's outgoing arrow indicates the events it publishes, while its incoming arrow indicates the events it subscribes to. The *TemperatureSensor* and *HumiditySensor* components publish `temperatureReading`- and `humidityReading` events, respectively. The *ComfortLevelMonitor*, on the other hand, is subscribed to these events and uses them to compute and log the comfort levels for each tent, and to decide how much the heating in each tent should be adjusted. To control the heating, it publishes `adjustHeating` events, to which the *HeatingController* components are subscribed. Whenever a *HeatingController* receives an `adjustHeating` event, it adjusts its associated heater accordingly. Next to these basic events, the components deployed within a tent also publish an `online` event carrying their component type, tent identifier (i.e., an identifier obtained by reading out a GPS sensor), component ID and MAC address. The *ComfortLevelMonitor* subscribes to these events as it should know which components are deployed in which tents to process their events[2] and to send the `adjustHeating` event to the *HeatingControllers* in the appropriate tent. In addition, publishing these events allows the components to be arbitrarily moved on the festival terrain. For instance, when a *HeatingController* component is moved to another tent, the *ComfortLevelMonitor* will know about its new location and will therefore no longer send it `adjustHeating`

[1] i.e., a WSN application that uses event-based component middleware.

[2] It is assumed that every event contains its originating component's MAC address and component ID. This way, any event can be related with an `online` event to obtain the originating tent.

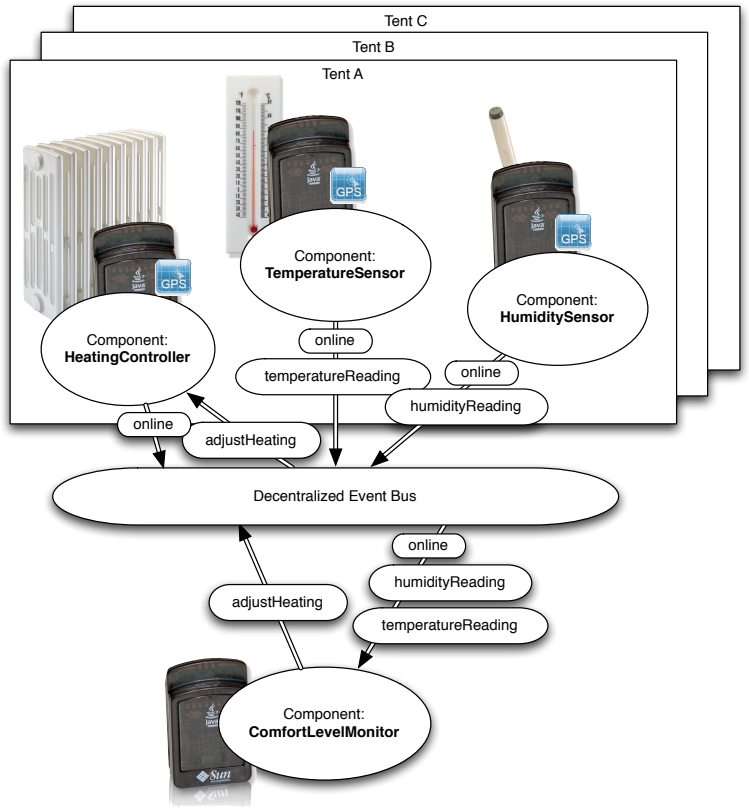


Figure 3.1: Overview of a Wireless Sensor Network application that controls the heating and logs the comfort levels of festival tents

events for the old tent. As shown in the figure, all the events are routed through the decentralized event bus offered by the event-based component middleware.

The functional requirements for this application are listed in Table 3.1. While functional requirements 1, 2, 3, and 5 are not that difficult to implement, the other requirements would clearly require to implement ad-hoc timers, state management and event-matching. For instance, to avoid that a heater keeps heating when the *ComfortLevelMonitor* fails or gets disconnected from the WSN (i.e., F.R.4), the *HeatingController* has to reset its associated heater if no `adjustHeating` event has been received for a certain amount of time. Functional requirements 6 and 7, on the other hand, require that events are stored in memory upon reception, and that they can later be consulted and related with other events. In addition, functional requirement 6 requires that events can be send to specific components, rather than being broadcasted to all subscribers. Finally, functional requirement 8 requires that previously stored events can be removed from memory when new events have been received.

Clearly, programming this application requires the programmer to deal with a fair amount of *accidental* complexity, which is inherent to using event-based WSN middleware. Therefore, the CrimeSPOT language is explicitly designed to deal with this *accidental* complexity and to allow the programmer to focus

	Description
F.R.1	The *HumiditySensor* and *TemperatureSensor* have to publish their sensor readings at set intervals.
F.R.2	The *HumiditySensor*, *TemperatureSensor* and *HeatingController* have to publish their online presence and location at set intervals.
F.R.3	The *HeatingController* has to adjust its associated heater according to a received `adjustHeating` event.
F.R.4	The *HeatingController* has to make sure that its associated heater won't keep heating when the *ComfortLevelMonitor* fails or gets disconnected from the WSN.
F.R.5	The *ComfortLevelMonitor* has to compute a tent's heating level based on a received `temperatureReading` event and publish this level in an `adjustHeating` event.
F.R.6	The *ComfortLevelMonitor* has to control the heating for each tent individually by sending `adjustHeating` events only to the *HeatingControllers* in the tent to be heated.
F.R.7	The *ComfortLevelMonitor* has to relate received `humidityReading` and `temperatureReading` events that originate from the same tent and use them to compute and log that tent's comfort level.
F.R.8	The *ComfortLevelMonitor* has to make sure that only the most recent sensor readings from a certain tent are used for computing the heating- and comfort levels.

Table 3.1: Functional requirements for a WSN application that controls the heating and logs the comfort levels of festival tents

on the application's *essential* complexity.

3.1.1 Programming Support Requirements

Because the running example is a representative event-based WSN application, we used its functional requirements to extract the requirements for a programming support solution that deals with the aforementioned *accidental* complexity. We categorized the results in *state management*- and *component interaction requirements*, which are listed in Table 3.2 and 3.3, respectively. Note that there's a clear separation between component interaction (i.e., publishing events) and invoking general application logic.

Every requirement enables one or more functional requirements of the running example. While most requirements are an obvious generalization of the running example's requirements, S.R.2 and S.R.3 are not. These requirements allow events to expire after a certain amount of time and allow programmers to react to such expirations. However, any solution meeting these requirements provides an elegant way for dealing with functional requirements such as F.R.4, which require timeouts. Later, we will further motivate the need for stored events to expire.

It must be noted that these programming support requirements also prove the *representativeness* of our running example. Analysis showed that various other WSN applications found in literature (cfr. Section 5.1) can be imple-

	Description	Enables
S.R.1	A component has to be able to store received events in its memory.	**F.R.4** **F.R.6** **F.R.7**
S.R.2	Events have to be able to expire after a certain amount of time, after which they are removed from the component's memory.	**F.R.4**
S.R.3	A component has to be able to react to the expiration of an event.	**F.R.4**
S.R.4	A component has to be able to consult previously received events.	**F.R.6** **F.R.7**
S.R.5	A component has to be able to relate events by comparing their payloads.	**F.R.6** **F.R.7**
S.R.6	A component has to be able to react to the reception of an event by removing older events from its memory.	**F.R.8**

Table 3.2: Programming Support Requirements concerning State Management

	Description	Enables
I.R.1	A component has to be able to invoke application logic at set intervals and to publish the results in events.	**F.R.1** **F.R.2**
I.R.2	A component has to be able to react to the reception of an event by invoking application logic.	**F.R.3** **F.R.7**
I.R.3	A component has to be able to react to the reception of an event by performing a computation and publishing its result in an event.	**F.R.5**
I.R.4	A component has to be able to publish events to specific components.	**F.R.6**

Table 3.3: Programming Support Requirements concerning Component Interactions

mented with a solution meeting these requirements. In the remaining part of this chapter, we will show how our solution, dubbed CrimeSPOT, meets all identified programming support requirements.

3.2 CrimeSPOT in a Nutshell

CrimeSPOT is a declarative, *rule-based* language to be used on top of event-based component middleware for wireless sensor networks. It provides programming support for building event-based WSN applications and is founded on the notions of *facts* and *interaction rules*.

To meet the state management requirements identified earlier, CrimeSPOT reifies received events as facts and stores them in a *Fact Base*. In general, facts can be added to the Fact Base (i.e., *asserted*), removed from the Fact Base (i.e., *retracted*), and they can *expire*. When a fact expires, the CrimeSPOT runtime will retract it from the Fact Base. In addition, a fact can also be *published*, which corresponds to the publication of an event. For a proper conversion between events and facts, the CrimeSPOT runtime can be configured in a declarative

way by adding event-to-fact mappings.

The interaction rules mainly support the interaction requirements. They can be used to specify, in a declarative manner, how a component should react when it receives a particular event or several related events. Within the conditions (or *body*) of an interaction rule, one can specify the events to react upon by specifying their reifying facts, while, in its *head*, one can specify the reaction (e.g., publishing a fact or invoking application logic). Additionally, interaction rules can also be used to specify that a component should publish a particular fact at certain intervals. For every component interaction, an interaction rule should be added to the CrimeSPOT runtime.

3.3 Architectural Overview

To give an idea of how CrimeSPOT is used, Figure 3.2 depicts the physical view on a WSN node running event-based component middleware. Note that the

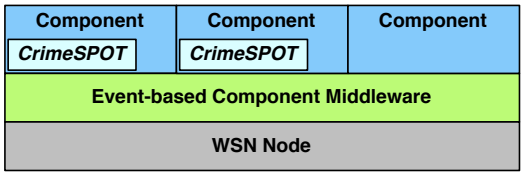


Figure 3.2: Physical view on a Wireless Sensor Network node running event-based component middleware

CrimeSPOT runtime is part of an event-based component and doesn't preclude such a component to interact with the underlying middleware. This is essential because CrimeSPOT is a building block for event-based components. Consequentially, *CrimeSPOT components* (i.e., components using CrimeSPOT) can run next to regular components without any problems.

3.3.1 CrimeSPOT Runtime

An architectural overview of the three-layered CrimeSPOT runtime is depicted in Figure 3.3.

The *infrastructure layer* binds the CrimeSPOT runtime to the underlying middleware. To this end, a middleware-specific implementation for the Middleware Bridge should be provided which transfers the events received from the event-based middleware to the Reification Engine. In addition, the Middleware Bridge should also implement all required middleware-specific functionality, which will be invoked by the Reification Engine (e.g., functionality to construct and publish an event). Clearly, the Middleware Bridge can be configured for any underlying event-based middleware.

The *reification layer* takes care of reifying events as facts upon reception (i.e., to be asserted in or retracted from the Fact Base), and reifying facts as events (i.e., to be published through the underlying middleware). To this end, the Reification Engine will use the event-fact mappings from its Configuration Base. As we will explain later, the Configuration Base can also contain

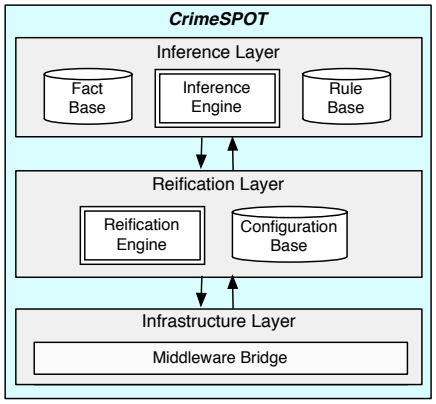


Figure 3.3: Architectural overview of the CrimeSPOT runtime

meta-facts, which further influence the Reification Engine's behavior. Event-fact mappings and meta-facts are application-specific and can be arbitrarily added to the CrimeSPOT runtime by a component. The cooperation of the Middleware Bridge and the Reification Engine is further illustrated in Figure 3.4. On the left-hand side, a fact is published and therefore reified as an event

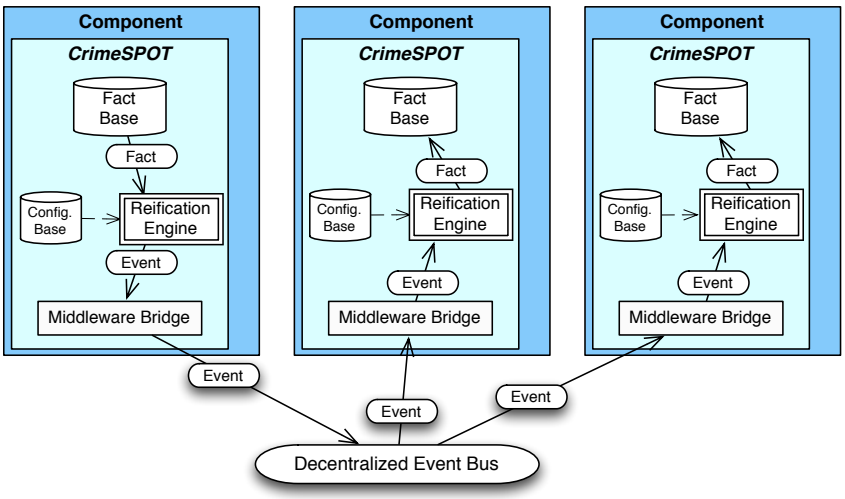


Figure 3.4: Fact publication and Event reification

by the Reification Engine and published to the event bus by the the Middleware Bridge. On the right-hand side, an event is received from the event bus and therefore transferred to the Reification Engine by the Middleware Bridge and reified as a fact and asserted in the Fact Base by the Reification Engine.

Next to the Fact Base, the *inference layer* of the CrimeSPOT runtime also contains the Rule Base, which stores the component's interaction rules. By evaluating the Rule Base against the Fact Base whenever the Fact Base is updated, the Inference Engine will *activate* the interaction rules as soon as their

conditions are met. Furthermore, because CrimeSPOT is a *truth maintenance system*, the Inference Engine will also *deactivate* the interaction rules as soon as their conditions are no longer met. The activation of an interaction rule corresponds to the invocation of its reaction (e.g., invoking application logic), while its deactivation corresponds to undoing its reaction (e.g., invoking compensating application logic).

3.3.2 Interacting with the CrimeSPOT Runtime

Figure 3.5 depicts the logical view on the possible interactions with the CrimeSPOT runtime. The interface of the CrimeSPOT runtime is simple. While all textual

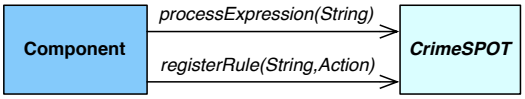


Figure 3.5: Logical view on the possible interactions with the CrimeSPOT runtime

expressions (e.g., facts, plain rules, etc.) can be added to the system through the `processExpression(String)` method, interaction rules invoking application logic can be added through the `registerRule(String,Action)` method.

3.3.3 Network View on Component Interactions

Interactions among CrimeSPOT components To illustrate the interactions between CrimeSPOT components, Figure 3.6 depicts a network-level view on two interacting CrimeSPOT components that don't have any event-fact mappings in their Configuration Base. The absence of these mappings is the default situation and therefore does not preclude two CrimeSPOT components to communicate. When a CrimeSPOT component publishes a fact that's not mapped to a particular event, the CrimeSPOT runtime will encode this fact in a CrimeSPOT event to which all CrimeSPOT components are subscribed. In the Figure, the component on the left-hand side publishes a fact

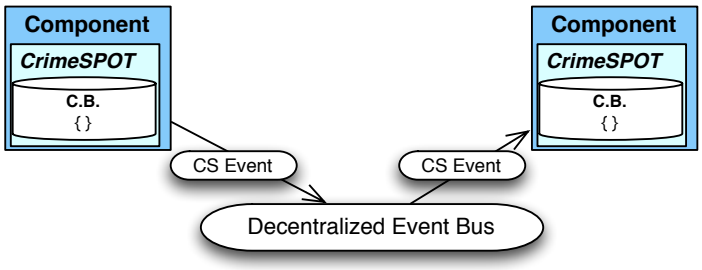


Figure 3.6: Two CrimeSPOT components exchanging CrimeSPOT events

that is reified as a CrimeSPOT event, published to the event bus and received by the component on the right-hand side. The receiving component will reify the CrimeSPOT event as the corresponding fact and assert it in its Fact Base.

Interactions with regular components As shown above, all CrimeSPOT components can interact in the absence of event-fact mappings. However, to interact with regular components, a CrimeSPOT component *should* add the required event-fact mappings to the CrimeSPOT runtime. The reason for this requirement is that regular components exchange events with a particular type and payload whose structure is agreed upon[3]. The purpose of an event-fact mapping is to map the structure of an event to the structure of a fact to allow proper reification. It will be used to reify received events as facts to be stored, and to reify facts as events to be published. For instance, if an event-fact mapping for temperature events is present in the Configuration Base, CrimeSPOT components can freely exchange these events with regular components. This is depicted in Figure 3.7, which shows the interactions between two CrimeSPOT components and a regular component. The CrimeSPOT component on the left-hand side publishes a temperature fact, which is reified as a temperature event and published to all components subscribing to this event. Conversely, as shown on the right-hand side, the CrimeSPOT components will use the same mapping to reify received temperature events as facts. As we will explain in Sec-

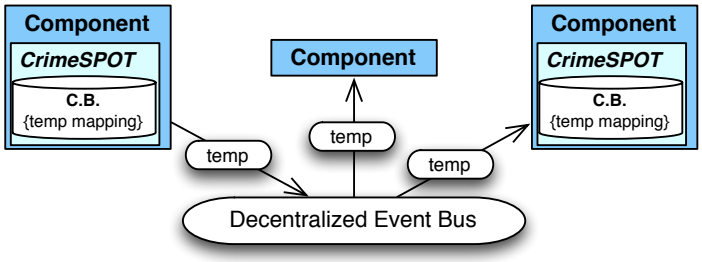


Figure 3.7: Event-based components exchanging `temp` events

tion 3.6.2, a component should add an event-fact mapping to the CrimeSPOT runtime for every event it subscribes to, and every event it plans to publish to regular components.

3.4 Inference Engine

The CrimeSPOT interaction rules have to be activated as soon as their conditions are met (i.e., as soon as the required events have been received, reified as facts and asserted in the Fact Base) and deactivated as soon as their conditions are no longer met (i.e., as soon as the required facts were retracted from the Fact Base). Consequentially, the Inference Engine has to re-evaluate all rules whenever the Fact Base is updated.

To obtain this functionality, the CrimeSPOT Inference Engine uses a *forward chaining* inference strategy. Forward chaining is a *data-driven* strategy which derives all valid conclusions given a fact- and a rule base. Note that the derivation of conclusions corresponds to the triggering of the interaction rules' reactions in the CrimeSPOT setting.

[3]As as example, one can consider an event with the `temperature` type and a payload containing one integer that represents the measured temperature in degrees Celsius.

As we will further explain in Appendix A, our forward chaining Inference Engine is based on the RETE algorithm [37], which allows *efficient* forward chaining. This is essential in the setting of wireless sensor networks, because WSN nodes only have low processing power available. The RETE algorithm caches its proofs and whenever the Fact Base is updated, it uses the update itself (e.g., the assertion of a fact) to derive only the new conclusions rather than attempting to derive all valid conclusions from scratch. This significantly reduces the performance overhead of Fact Base updates.

The original RETE algorithm was extended with various domain-specific features. For instance, the Inference Engine can be triggered at specific intervals and facts can expire after a certain amount of time, after which the Inference Engine will retract them from the Fact Base. In general, our Inference Engine is triggered:

- when a fact is asserted in the Fact Base,
- when a fact is retracted from the Fact Base (e.g., upon its expiration),
- and at specific intervals, specified by the programmer.

3.5 Grammar

The CRIMESPOT BNF grammar is depicted in Figure 3.8. We extended the Backus Naur Form notation with large square brackets to depict optional items and large parenthesis suffixed with an asterisk to depict zero or more repetitions. As mentioned earlier, every valid expression can be added to the CRIMESPOT runtime through the `processExpression(String)` method. In the following sections, we will introduce all language features shown in the grammar.

3.6 Facts

Our approach is founded on the notion of a fact rather than an event. Even though both notions look similar, they are significantly different from a semantic point of view [38]. While events are *transient*, typically only valid at the time of reception, facts are *persistent* (i.e., asserted in a Fact Base) and thus remain valid until they are explicitly removed. To blend both notions, we allow associating an expiration time with a fact.

Every fact has a *type* and optional *attributes* (i.e., name-value pairs, separated by the `=` symbol). Additionally, a fact can also carry *meta-attributes* representing its meta-data. A fixed number of meta-attributes can be added to a fact by listing them after the `@` symbol. An example is depicted in Listing 3.1.

```
1 temperatureReading(Celsius = 27,
2                    Fahrenheit = 81)@[factExpires(Seconds=600)].
```

Listing 3.1: A temperatureReading fact

The fact represents a temperature reading of 27 degrees Celsius or 81 degrees Fahrenheit, which remains valid for ten minutes. When adding this fact to the CRIMESPOT runtime, it will be asserted in the Fact Base and the Inference Engine will read the `factExpires` meta-attribute and consequentially retract the fact after ten minutes.

3.6.1 Controlling a Fact's Scope

Next to the `factExpires` meta-attribute, a fact can also carry the `to` meta-attribute, which can be used to publish the fact to a given destination and thereby control its scope. Within this meta-attribute, the fact's destination should be specified by providing values for the *ID*- and/or the *MAC* attribute, which refer to the destination component's ID and sensor node MAC address, respectively. To broadcast a fact to all sensor nodes, or to all components running on a particular sensor node, the `any` keyword can be used as a value for the *MAC*- or *ID* attribute, respectively. Listing 3.2 illustrates the use of the `to` meta-attribute.

```
1 temperatureReading(Celsius = 27)@[to(MAC=any)],
2                                  factExpires(Seconds=600)].
3 temperatureReading(Celsius = 27)@[to(MAC="1234:1234:1234:1234",ID=3)],
4                                  factExpires(Seconds=600)].
```

Listing 3.2: Publishing a temperatureReading fact

When adding these facts to the CrimeSPOT runtime, they will be asserted in the Fact Base and the fact on line 1 will be broadcasted to all CrimeSPOT components, while the fact on line 3 will be send only to the specifically addressed component. If the underlying middleware allows directed event publications, the Middleware Bridge can consult the values from the *to* meta-attribute to employ this feature. If not, it can consult these values upon event reception and misaddressed events can be dropped. As discussed earlier, every component that receives a fact will assert it in its Fact Base. Additionally, the component's Inference Engine will also read the `factExpires` meta-attribute, if present, to deal with its expiration.

3.6.2 Mapping Events to Facts

As shown above, a fact is self-explaining. It has a particular type and all of its attributes are named. An event, on the other hand, is not. Next to its type, it has a particular payload whose structure is *agreed upon* by the components exchanging the event. The payload consists of one or more numbered values (e.g., integers or strings), but only the components exchanging the event know the semantics of each value. For instance, components can exchange an event of type 101 with a payload containing two integers and agree that this event represents a temperature reading measured in degrees Celsius (i.e., the first value) and Fahrenheit (i.e., the second value), which remains valid for ten minutes. To reify such events as facts and add them to the Fact Base, the structure agreement should be made explicit by the programmer. This can be done by adding event-fact *mappings* to the CrimeSPOT runtime.

To map the aforementioned temperature event to a `temperatureReading` fact with `Celsius` and `Fahrenheit` attributes, the programmer can use the mapping depicted in Listing 3.3.

```
1 mapping temperatureReading(Fahrenheit=?tempf,
2                            Celsius=?tempc)@[factExpires(Seconds=600)]
3    <=> Event_101(Integer=?tempc, Integer=?tempf).
```

Listing 3.3: Mapping a temperature event to a temperatureReading fact

A mapping starts with the `mapping` keyword, followed by the reifying fact and a description of the event which are separated by the `<=>` symbol. The event description, on the right-hand side of the `<=>`, resembles the fact notation. It has a *type* and one or more *fields*, which are comparable to attributes but have a *type* rather than a name. Every field represents a value from the payload: its type represents the value's type and its position in the event description represents the value's position in the payload. The corresponding values of the fact's attributes and the event's fields are related by using identical variables in the mapping (e.g `?tempc` and `?tempf` in the example). Note that, unlike the fields of an event, the attributes of a fact are not ordered, but uniquely identified by their name.

After adding event-fact mappings to the CrimeSPOT runtime, they will be stored in the Configuration Base and used by the Reification Engine when processing incoming events (i.e., to reify them as the corresponding fact) and when publishing facts (i.e., to reify them as the corresponding event). In addition, the Middleware Bridge can subscribe the component to the mapped event type as soon as the mapping is added. As a result, mappings allow a CrimeSPOT component to communicate freely with regular components.

3.6.2.1 Event Meta-Data

Next to a payload, most event-based component middleware also associate *meta-data* with an event. For instance, an event can carry, in addition to its payload, its originating component's ID and sensor node MAC address. To avoid losing this data upon event reification, CrimeSPOT also incorporates it in its facts using the *meta-attributes*. For every event meta-datum, the Middleware Bridge can add a meta-attribute to the fact reifying the event. In general, it is assumed that a `from(ID=..,MAC=..)` meta-attribute specifying the fact's origin is present for every fact in the Fact Base.

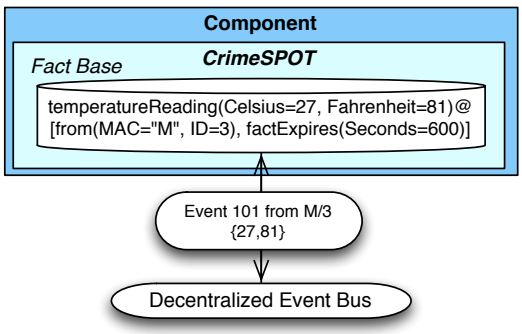


Figure 3.9: Illustrating the temperature event mapping

Figure 3.9 further illustrates the meaning of the mapping in Listing 3.3. When read from top to bottom, the Figure shows how the `temperatureReading` fact is reified as an event upon its publication, and conversely, when read from bottom to top, the Figure shows how the temperature reading event is reified as a `temperatureReading` fact upon its reception.

3.7 Interaction Rules

As mentioned before, interaction rules have to be used for specifying a component's interactions. An interaction rule consists of a *body* containing one or more conditions, and a *head* containing either a fact or an action (i.e., a Java object encapsulating application logic). The head and the body are separated by the arrow `<-` symbol.

⟨*rule*⟩ ::= ⟨*fact*⟩ `<-` ⟨*body*⟩

⟨*body*⟩ ::= ⟨*condition*⟩ (`,` ⟨*condition*⟩)*

⟨*condition*⟩ ::= ⟨*fact*⟩
| ⟨*extralogical_condition*⟩

Figure 3.10: Syntax for an interaction rule with a fact in its head

As we will extensively explain in the following sections, a condition can be either *logical* or *extra-logical*. A logical condition specifies a fact and obtains a *match* for every matching fact in the Fact Base. An extra-logical condition, on the other hand, uses a built-in operator to obtain matches. As soon as all conditions in the body of a rule have a match, the rule itself has a match and will be *activated* for this match by the Inference Engine. Depending on the rule's head, its activation results in either asserting a fact in the Fact Base, which can optionally be published, or invoking an action's application logic.

Every time the Fact Base is updated (e.g., when a new event is received from the event bus or when a fact expires), all rules are re-evaluated against the Fact Base (i.e., conceptually). Rules of which a single condition did not yet have a match, can obtain a match upon the next evaluation against the Fact Base. Conversely, a rule can also lose a match if one of its conditions lost a match. As soon as a rule loses a match, the Inference Engine will *deactivate* it for this match. Depending on the rule's head, its deactivation results in either retracting a fact from the Fact Base, or invoking an action's application logic.

The above can be summarized by the invariant that, at any time, an interaction rule is *activated* for each of its matches. As an example, consider the rule in Listing 3.4, which asserts a `temperature` fact for every `temperatureReading` fact in the Fact Base.

```
1 temperature(Celsius=?temp)
2    <- temperatureReading(Celsius=?temp).
```

Listing 3.4: An interaction rule for copying temperatureReading facts to temperature facts

If the Fact Base contains three matching `temperatureReading` facts (e.g., with `?temp` 20, 20 and 24), this rule is activated three times with a corresponding binding for `?temp` and the three corresponding `temperature` facts will consequentially be present in the Fact Base. As soon as a new `temperatureReading` fact is asserted in the Fact Base, the rule obtains a new match and hence, it will be *activated* for this match and the corresponding `temperature` fact will be asserted. Conversely, as soon as one of the `temperatureReading` facts is retracted from the Fact Base, one of the rule's matches is lost and hence, the

rule will be *deactivated* for that match and the corresponding `temperature` fact will be retracted.

More details concerning the interaction rules' behavior will be given in Section 3.7.2. In the next section, we will first elaborate on the conditions that can be used in an interaction rule's body.

3.7.1 Conditions

Typically, a condition is used for matching facts from the Fact Base. To this end, one can employ a *logical condition* which specifies the facts to be matched. For instance, if a component should react to the reception of a temperature reading, one of the following conditions can be used in the body of its interaction rule:

```
1 temperatureReading(Fahrenheit=?temp)
2 temperatureReading(Celsius=?temp)
3 temperatureReading(Celsius=40)
4 temperatureReading(Celsius<40)
```

Since the first two conditions employ a variable for the measured temperature, the measured value is not constrained. Any `temperatureReading` with a *Fahrenheit* attribute will provide a match for the first condition, while any `temperatureReading` with a *Celsius* attribute will provide a match for the second condition. The third condition is further constrained as it will only be matched when the received temperature reading is 40 degrees Celsius. The fourth condition employs the `<` operator and hence will only be matched when the received temperature reading is less than 40 degrees Celsius.

Note that it suffices to only specify the attributes required to do the matching within a logical condition. A `temperatureReading` fact with both *Celsius* and *Fahrenheit* attributes can also match the previous conditions.

3.7.1.1 Incorporating Meta-Data

As mentioned earlier, every fact carries meta-attributes next to its regular attributes. For all event meta-data made available by the underlying event-based middleware, such meta-attributes can be added to the facts reifying the events. To incorporate this meta-data in the interaction rules, the meta-attributes can also be matched in a condition. For instance, if a component should react to the reception of a temperature request, the following condition can be used in the body of its interaction rule:

```
1 getTemperature()@[from(MAC=?mac, ID=?id)]
```

For every match, the variables `?id` and `?mac` will be bound to the requesting component's ID and sensor node MAC address, respectively. Since all variables bound in the body of an interaction rule are available in its head, these variables' values can be used in this example to answer the request by publishing the current temperature to the requesting component.

The `this.MAC` or `this.ID` keyword can be used as a value for the *MAC* or *ID* attribute within the *from* meta-attribute. While the `this.ID` keyword substitutes for the current component's ID, the `this.MAC` keyword substitutes for the current sensor node's MAC address. For instance, in the context of our running example, a component that publishes `online` facts can use the following condition to match the `online` fact that it has published before. A match for this condition will bind `?tent` to the identifier of the current tent.

```
1 online(Tent=?tent)@[from(MAC=this.MAC, ID=this.ID)]
```

3.7.1.2 Relating Facts

Wireless sensor network nodes often have to react to the occurrence of related events. For instance, as the *ComfortLevelMonitor* from the running example should log the comfort levels in each tent, it should relate humidity- and temperature readings which originate from the same tent to compute the associated comfort level and do the logging. For this purpose, several conditions in an interaction rule's body can be related by using identical variables. Because, within a rule' match, a variable can be bound to only one value, all occurrences of the same variable have to be consistent. This behavior is also called *unification*. As an example, we can consider the *ComfortLevelMonitor* component's interaction rules for logging the comfort levels. These interaction rules relate facts based on their publisher and based on their contents.

Publisher-based relations As mentioned before, the meta-attributes of a fact can be matched within a logical condition. Consequentially, two preprocessing interaction rules can be added to the *ComfortLevelMonitor* to relate the received sensor readings with their originating tent. The following rule relates `humidityReading` facts with `online` facts and stores the result in `humidityInTent` facts.

```
1 humidityInTent(Percent=?humid, Tent=?tent)
2    <- humidityReading(Percent = ?humid)@[from(MAC=?hMac, ID=?hId)],
3       online(Tent = ?tent)@[from(MAC = ?hMac, ID = ?hId)].
```

Listing 3.5: Relating humidity readings with online facts

The body of an interaction rule specifies a conjunction and all conditions are separated by a comma. Consequentially, this rule requires `humidityReading`- and `online` facts to be available in the Fact Base before it will be triggered. However, it also requires these facts to be related. If a `humidityReading` is received, it will match the first condition. The variable `?humid` will be bound to the measured humidity and the variables `?hMac` and `?hId` will be bound to the publishing component's MAC address and ID, respectively. As a consequence, only the `online` fact published by the very same component will match the second condition and will bind `?tent` to the originating tent. As soon as both conditions are matched and the variable bindings are consistent, the rule is activated and the `humidityInTent` fact (with `?humid` and `?tent` substituted by their values) will be asserted in the Fact Base. Analogously, a rule for relating `temperatureReading` facts with `online` facts in `temperatureInTent` facts can be added to the *ComfortLevelMonitor*.

Content-based relations The facts asserted by the aforementioned preprocessing rules can now be related in the interaction rule for logging the comfort levels. Note that this rule has to invoke application logic when it's activated and that it therefore requires an *action* (i.e., `logAction`) rather than a fact in its head. We will further explain the use of actions in Section 3.7.2.

```
1 logAction
2    <- humidityInTent(Percent=?humid, Tent=?tent),
3       temperatureInTent(Celsius=?temp, Tent=?tent).
```

As soon as a `humidityInTent` fact is asserted in the Fact Base, it will match the first condition and the variable `?tent` will be bound to the originating tent. Consequentially, only a `temperatureInTent` fact for the same tent `?tent` will provide a valid match for the second condition. When the rule is activated, the application logic in the `logAction` will be invoked to compute and log the tent's comfort level.

It must be noted that an interaction rule with a body consisting of multiple conditions does not require these conditions to be matched in any particular order. A match for such an interaction rule only requires a match for each condition with consistent variable bindings.

However, in general, care must be taken to make sure that a condition matches only the expected amount of times. To illustrate this, we can reconsider the running example. In this application, it is perfectly possible that a certain humidity sensor node is reset due to some failure. If this happens, the node will restart and re-publish its `online` fact, and as a result, the *ComfortLevelMonitor*'s Fact Base will contain two such facts[4]. Consequentially, the rule for asserting `HumidityInTent` facts discussed above will have two matches for the condition on line 3, and will thus be activated twice for each for each humidity reading from the reseted node's tent. Figure 3.11 further illustrates this issue.

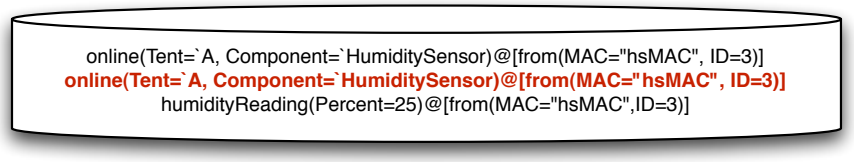


Figure 3.11: Faulty matching illustrated

The `humidityReading` in the Fact Base will match the condition on line 2 and bind `?hMac` and `?hId`. However, because two `online` facts with consistent values for these variables will match the second condition, the rule itself will have two matches and will consequentially be activated twice. This results in the assertion of two `humidityInTent` facts for a single `humidityReading` fact. To allow the programmer to deal with this unintended behavior, CrimeSPOT offers functionality to retract old facts when new facts are received. In this case, this would be useful to retract a component's existing `online` fact when a new one is received. We will explain the use of this functionality in Section 3.8.

3.7.1.3 Fine-grained Matching Control

When matching facts using logical conditions, it might be desired to get more control over the matching process. To this end, our language offers some built-in

[4]That is, until the oldest one expires.

meta-attributes that can be added to a fact specification in a condition. Unlike the meta-attributes specifying a fact's associated meta-data, these built-in meta-attributes are not part of the facts in the Fact Base, but allow the programmer to exert control over the matching process for the associated fact specification. The built-in meta-attributes can be combined as desired.

Match Expiration In certain cases, it is useful to make a match for a condition expire after a specific amount of time. For instance, when dealing with `temperatureReading` facts, a more recent temperature reading can have a different meaning than an older temperature reading. This can be specified using the *matchExpires* meta-attribute. In case the programmer only wants temperature readings with a maximum age of ten seconds to match an interaction rule's condition, she can express this using the condition from Listing 3.6.

```
1 temperatureReading(Celsius=?temp)@[from(MAC=?m, ID=?i),
2                                    matchExpires(Seconds=10)]
```

Listing 3.6: A condition for matching only recent temperature readings

Any fact matching a condition with an associated *matchExpires* meta-attribute will only match this condition for the specified amount of time. Note that the expiration of the match does not imply the expiration of the matched fact. Hence, nothing precludes other rules to use such a fact for matching their conditions. Figure 3.12 further illustrates the impact of the *matchExpires* meta-attribute on the above condition. At every point in time (denoted in seconds), the contents of the Fact Base and the matches for the condition are shown. Temperature readings are received at 0s and 4s. For simplicity, these temperature readings are assumed not to expire. However, due to the *matchExpires* meta-attribute, their match for the above condition does expire after ten seconds. Hence, while the facts remain in the Fact Base and therefore can still match other conditions, they do not match this condition for more than ten seconds.

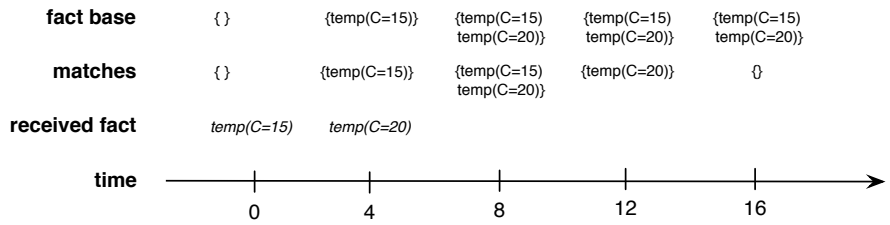


Figure 3.12: Illustrating Match Expiration

Match Verification Additionally, it is also useful to specify the expected matching for a condition, for instance, to do some sort of error handling whenever the expected matching is not obtained. To this end, the *matchEvery* meta-attribute can be added to a fact specification. Whenever a condition carrying this meta-attribute isn't matched in the specified amount of time, a `timedOut` fact will be asserted in the Fact Base. Such a `timedOut` fact specifies exactly which condition was violated by specifying the rule in which the condition was

used[5], and the type of the fact specification carrying the *matchEvery* meta-attribute. As an example, we can consider the rule from Listing 3.7.

```
1 gotReadingFrom(MAC=?m)
2    <- temperatureReading(Celsius=?temp)@[from(MAC=?m, ID=?i),
3                                          matchEvery(Seconds=60)].
```

Listing 3.7: A rule with a condition that expects a match at least every minute

As soon as this condition doesn't get a new match within one minute, the following `timedOut` fact will be asserted in the Fact Base.

```
1 timedOut(Head=‘gotReadingFrom_1, ViolatedCondition=‘temperatureReading).
```

To do the appropriate error handling, this fact can be matched in the condition of another rule to signal the malfunctioning of the sensor node publishing the temperature readings. As soon as the condition obtains a new match, the corresponding `timedOut` fact is retracted from the Fact Base. Figure 3.13 depicts the impact of the *matchEvery* meta-attribute. At every point in time (denoted in seconds), the contents of the Fact Base and the matches for the condition are shown[6]. Temperature readings are expected to be received every 60 seconds, but are for some reason received at 0s and 90s. Due to the *matchEvery* meta-attribute, a `timedOut` fact is asserted at 60s because the expected matching was not obtained. Afterwards, at 90s, a new temperature reading is received and the `timedOut` fact is consequentially retracted.

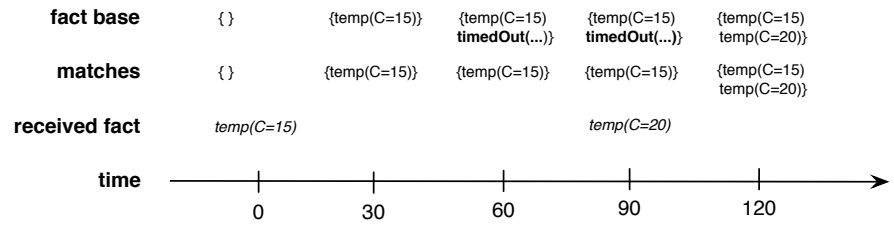


Figure 3.13: Illustrating Match Verification

3.7.1.4 Extra-logical Conditions

Up till now, we have only considered using logical conditions to match facts from the Fact Base and provide variable bindings. While logical conditions are often sufficient to specify a reaction to incoming events, it is also desirable to bind variables using other sources (e.g., method invocations), or to add additional constraints to the body of an interaction rule. To this end, our language provides the programmer with some extra-logical conditions, which we will briefly discuss in this section.

Incorporating component state The state of the component using CrimeSPOT can be accessed from within the rules' bodies using the built-in *is*-operator. This operator expects either a constant value or a variable on its left-hand side and

[5]Every interaction rule is uniquely identified by the fact type of the fact in its head, or the keyword `actionRule` if the head contains an action, followed by its lexicographical number.

[6]For simplicity, the `gotReadingFrom` facts are not shown in the Fact Base.

a method invocation on its right-hand side. It only provides a match if the expression on the left-hand side can be unified with the result of the method invocation. In addition, the arguments for the method to invoke should be either constants, or variables that were bound by the previous conditions in the rule's body. For instance, one can bind a variable with the current temperature by invoking one of the component's methods as shown below.

```
1 ?temp is getTemperature()
```

Another relevant use is to compute a comfort level based on the temperature- and humidity readings originating from the same sensor node. In that case, the following body can be used.

```
1 humidityReading(Percent = ?h)@[from(MAC=?mac)],
2 temperatureReading(Celsius = ?t)@[from(MAC=?mac)],
3 ?comfortLevel is computeComfortLevel((Number)?h, (Number)?t)
```

Note that variable method arguments have to be type-casted to their values' Java type[7]. For this method invocation to succeed, the component should have a method with the following signature.

```
1 Attribute computeComfortLevel(Number, Number);
```

In general, a method is invoked for every match for the previous conditions in the rule's body. For instance, in the previous example, *computeComfortLevel* is invoked for every pair of humidity- and temperature readings originating from the same sensor node. In case there are no preceding conditions, the method is invoked only once. However, the programmer can also specify an invocation schedule by adding a scheduling option to the method invocation. Scheduling options employ the same notation as meta-attributes. To illustrate their use, we can reconsider the sensing components from our running example, which have to publish their sensor readings at set intervals. An interaction rule specifying this behavior for the *HumiditySensor* component is shown in Listing 3.8.

```
1 humidityReading(Percent = ?p)@[to(MAC=any),
2                                factExpires(Seconds = 600)]
3    <- ?p is getHumidity()@[renewEvery(Seconds = 600)].
```

Listing 3.8: Publishing humidity readings using the is operator

The scheduling option added to the method invocation specifies that, whenever the method is invoked, it should be re-invoked after ten minutes.

Either the *renewEvery*- or the *evalEvery* option can be used to schedule additional invocations. While *renewEvery* invalidates the previous match for the *is*-operator, *evalEvery* keeps the older matches valid. The latter can, for instance, be used to keep a history of humidity readings in the Fact Base, as shown in Listing 3.9.

```
1 humidityReading(Percent = ?p)
2    <- ?p is getHumidity()@[evalEvery(Seconds = 600)].
```

Listing 3.9: Keeping a history of humidity readings

Every ten minutes, a new `humidityReading` will be asserted in the Fact Base. Using *renewEvery*, on the other hand, would replace the asserted humidity reading every ten minutes. This can be observed in Figure 3.14, which depicts the

[7]The most commonly used Java types are `Number`, `Fraction`, `Symbol`, `Text` and `List`, which all subclass `Attribute`.

matches for the *is*-operator together with the resulting Fact Base at every point in time (denoted in minutes).

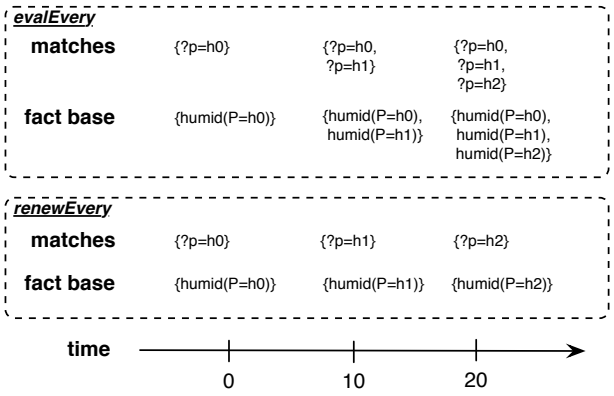


Figure 3.14: Scheduling method invocations: evalEvery vs. renewEvery

Other useful constructs Finally, four other constructs can be used as a condition in a rule's body. An *External Constraint* can be used for specifying the required relationship between variables, or between variables and constants. To this end, every relational operator is built-in and expects a variable on its left-hand side, and a variable or constant on its right-hand side. It provides a match if the specified relationship holds. For instance, to react to the occurrence of two temperature readings where one temperature reading is larger than the other, the following body can be used.

```
1 temperatureReading(Celsius=?t1),
2 temperatureReading(Celsius=?t2),
3 ?t1 > ?t2
```

In contrast to external constraints, *internal constraints* can be used within a fact specification. For instance, if the value of the second `temperatureReading` is irrelevant to the interaction rule, the above body can be rewritten as shown below.

```
1 temperatureReading(Celsius=?t1),
2 temperatureReading(Celsius<?t1)
```

The *Negation* can be used to check for the absence of matching facts. To this end, a logical condition can be preceded by the `not` operator, which only provides a match if the logical condition is unmatched. For instance, to check for the absence of an `adjustHeating` fact, the following condition can be employed.

```
1 not adjustHeating(Level=?l)
```

The *findall* operator can be used to accumulate values from several facts in a `List`. As its second argument, it takes a list of one or more conditions (i.e., its body), which bind certain variables in every match[8]. In addition, the variable whose bindings should be accumulated in the list should be given as findall's first argument, and the variable to be bound to the resulting list should

[8]Note that this body is also conceptually re-evaluated whenever the Fact Base changes.

be given as findall's last argument. At any time, the findall operator provides a match which binds this last variable to the list of accumulated values. As an example, we can consider the body of an interaction rule residing in a controller for smoke detectors. If all smoke detectors publish a smoke event upon smoke detection, and the controller should sound the alarm as soon as three different smoke events have been received, the body from Listing 3.10 can be used in the interaction rule.

```
1 findall(?m,
2         [smoke()@[from(MAC=?m)]],
3         ?alarmingSensors),
4 length(?alarmingSensors, ?l),
5 ?l >= 3
```

Listing 3.10: Collecting smoke facts using findall

At all times, `?alarmingSensors` will be bound to a list containing the publishers of the `smoke` facts in the Fact Base. However, the interaction rule will only be activated when this list contains at least three values. This example also illustrates the use of the *length* operator, which can be used to compute the length of a given list.

3.7.2 Reactions

Having discussed the kinds of conditions that can be used in the body of an interaction rule, we shift our focus to the head. The content of the head (i.e., either a *fact* or an *action*) determines the behavior that corresponds to the triggering of an interaction rule, or indirectly, it determines the reaction to the reception of events. In general, two reactions can occur: a fact can be asserted in or retracted from the Fact Base, or application logic can be invoked.

3.7.2.1 Fact Assertion or Retraction

If an interaction rule's head contains a fact, this fact will be asserted in the Fact Base for every activation of the rule. On the other hand, when the rule is deactivated, the corresponding fact will be retracted from the Fact Base. Hence, the view a component has on its world is always up-to-date.

As every match for a rule provides bindings for the variables that occur in its body, these variables can be used in the head. For instance, the previously discussed rule for keeping a history of humidity readings in the Fact Base (cfr. Listing 3.9) asserts `humidityReading(Percent=?p)` facts in which the variable `?p` is substituted for its value.

Publishing facts A common reaction to the reception of facts is to publish another fact. As discussed earlier, facts can be published on the underlying middleware's event-bus by giving it a *to* meta-attribute that specifies its destination. For instance, the following interaction rule can be added to the CrimeSPOT runtime to answer temperature requests.

```
1 temperatureReading(Celsius=?temp)@[to(MAC=?mac, ID=?id)]
2    <- getTemperature()@[from(MAC=?mac, ID=?id)],
3       ?temp is getTemperature().
```

Whenever a `getTemperature` fact is received, the *getTemperature()* method will be invoked and the `temperatureReading` fact will be published to the requester.

Note that, when an interaction rule is deactivated, its published fact should also be retracted in the remote Fact Bases. To this end, retractions are also published through the underlying middleware[9].

Fact Expiration It is often desired to retract a fact from the Fact Base before the interaction rule that asserted it is deactivated. To this end, the programmer can add a *factExpires* meta-attribute to the fact in which its lifetime can be expressed. Such fact expiration was, for instance, specified in the interaction rule for publishing `humidityReading` facts:

```
1 humidityReading(Percent = ?p)@[to(MAC=any),
2                                factExpires(Seconds = 600)]
3    <- ?p is getHumidity()@[renewEvery(Seconds = 600)].
```

The `humidityReading` fact expires after ten minutes and will consequentially be retracted from the local and remote Fact Bases. As such, this rule, which is re-activated every ten minutes, does not require a retraction for the humidity reading to be published.

3.7.2.2 Invoking Application Logic

Interaction rules that have an *action* as their head will invoke application logic whenever they are activated or deactivated. To illustrate this, we can consider the interaction rule for logging the comfort levels in our running example. This rule can be added to the *ComfortLevelMonitor* component's Rule Base using the `registerRule(String,Action)` method as shown in Listing 3.11. The `registerRule` method takes the body of the interaction rule as its first argument and an instance of the abstract `Action` type as its second argument. To specify the application logic to invoke, the instance of the `Action` type should provide an implementation for the `activated(TypedObject)` and `deactivate(TypedObject)` methods. These methods will be invoked whenever the interaction rule is activated and deactivated, respectively.

```
1 registerRule(
2    "humidityInTent(Percent=?humid, Tent=?tent), " +
3    "temperatureInTent(Celsius=?temp, Tent=?tent)",
4    new Action() {
5      public void activated(TypedObject args) {
6         // Compute and log the comfort level using
7         // args.getValue("humid"),  args.getValue("temp"),
8         // and args.getValue("tent")
9      }
10     public void deactivate(TypedObject args) {
11        // no compensating action for this rule
12     }
13 });
```

Listing 3.11: Adding an interaction rule for invoking application logic to the CrimeSPOT runtime

Whenever the interaction rule is triggered, the `TypedObject` instance passed to both of these methods can be used to consult the bindings for the variables that

[9] In the example, it is assumed that `getTemperature` requests remain in the Fact Base, and hence, that the interaction rule will never be deactivated.

occur in the body of the interaction rule. For instance, the binding for `?temp` can be obtained by invoking `getValue("temp")` on the `TypedObject`.

3.7.3 Precedence

In CrimeSPOT, the precedence between the interaction rules is determined by the order in which they were added to the CrimeSPOT runtime. If multiple rules employ a condition that is matched by a newly asserted fact, the rule that was added first will be activated first. Conversely, when multiple rules employ a condition that is no longer matched due to the retraction of a fact, the same precedence is retained for their deactivation.

3.8 Fine-tuning the Reification Engine

Since all facts in the Fact Base are taken into account for triggering the interaction rules, it's important to keep only the relevant ones. Up till now, we've discussed that facts can *expire* and are consequentially retracted from the Fact Base. However, the expiration of facts is not sufficient to keep the Fact Base clean. Often, a newly received fact *subsumes* older facts which should consequentially be retracted. In addition, it can occur that an incoming fact should be *dropped* right away rather than being asserted in the Fact Base.

In CrimeSPOT, these concerns can be expressed through *meta-facts*, which can be added to the CrimeSPOT runtime just like any other textual expression. While meta-facts look like regular facts, they represent the configuration of the Reification Engine and are stored in the Configuration Base rather than the Fact Base. Whenever the Reification Engine processes an incoming event, it will act according to its meta-facts. Figure 3.15 depicts the event reification pipeline. When the Reification Engine receives an event, it will first reify it as a fact and verify whether it should be dropped. If the fact shouldn't be dropped, the Reification Engine will retract all facts that were subsumed by the newly received fact, after which it will assert the new fact in the Fact Base.

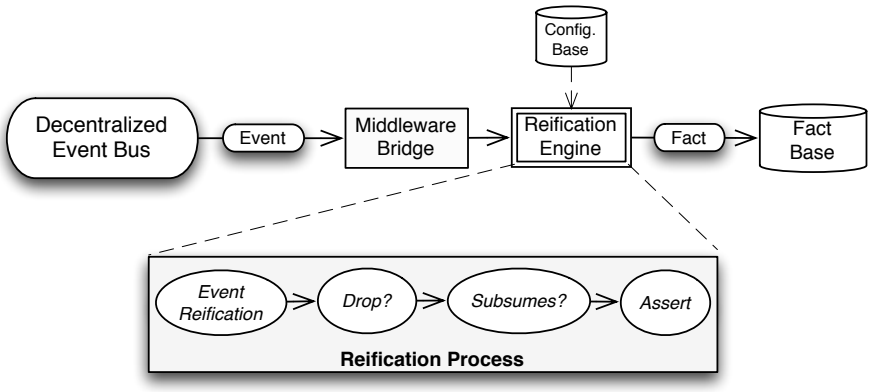


Figure 3.15: Event reification pipeline

3.8.1 Specifying Subsumption Strategies

The subsumption meta-fact can be used to specify exactly which facts should be removed from the Fact Base when a new fact is received. For instance, for the *HeatingController* component from our running example, only the last `adjustHeating` fact is of relevance. This fact will be used for steering the component's associated heater and no history of heating levels should be kept. Hence, before asserting a new `adjustHeating` fact, the previous ones should be removed. In other words, every new `adjustHeating` fact *subsumes* the previously stored one. This subsumption strategy can be expressed by adding the subsumption meta-fact from Listing 3.12 to the CrimeSPOT runtime.

```
1 subsumes!(Incoming=adjustHeating(Level = ?new),
2           Fact=adjustHeating(Level = ?old)).
```

Listing 3.12: Specifying a subsumption strategy for adjustHeating facts

While the *Incoming* attribute specifies the newly reified event, the *Fact* attribute specifies which facts should be retracted from the Fact Base before the new fact is asserted (i.e., all matching facts[10]). Because the variables specifying the heating level in either attribute are distinct, their values aren't required to be equal and hence every `adjustHeating` fact is retracted from the Fact Base when a new one is received. The impact of this meta-fact on the Fact Base is further illustrated in Figure 3.16. At every point in time, the contents of the Fact Base are shown together with the received `adjustHeating` fact and the set of facts subsumed by this new fact. Note that, every time a new `adjustHeating` fact is received, the previously stored one is retracted due to the presence of the above subsumption meta-fact in the Reification Engine's Configuration Base.

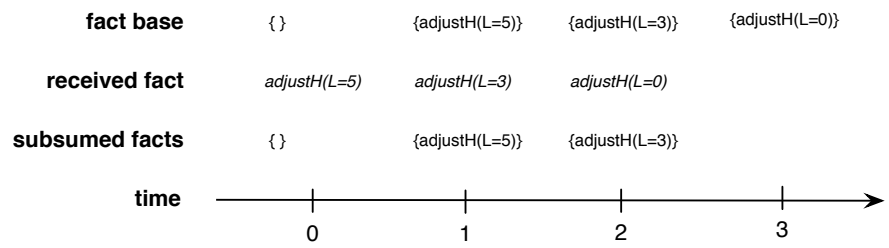


Figure 3.16: Subsumption impact for adjustHeating facts

By using the same variables in both fact specifications, more specific subsumption strategies can be expressed as the values bound to these variables should be equal for the subsumption to take place. This is useful, for instance, to deal with the unintended matching behavior in the *ComfortLevelMonitor* component from the running example (cfr. Figure 3.11). At any time, the *ComfortLevelMonitor* should only have a single `online` fact in its Fact Base for every running component. To avoid the faulty matching when a running component fails, restarts, and republishes its `online` fact, the following subsumption meta-fact should be added to the CrimeSPOT runtime:

[10] In case a certain incoming fact should subsume multiple fact types, multiple subsumption meta-facts should be added for this fact.

```
1 subsumes!(Incoming=online(Tent = ?new, Component = ?comp)@[from(MAC=?mac)],
2           Fact=online(Tent = ?old, Component = ?comp)@[from(MAC=?mac)]).
```

Listing 3.13: Specifying a subsumption strategy for online facts

When a new `online` fact is received and the above meta-fact is processed by the Reification Engine, `?mac` and `?comp` will be bound to the new fact's originating MAC address and component name, respectively. Hence, only the existing `online` fact originating from the very same component will match the *Fact* specification and will consequentially be retracted from the Fact Base before the new `online` fact is asserted. Note that nothing precludes a component to restart in another tent because the variables specifying the tent identifiers are distinct.

Finally, a subsumption meta-fact can also have a *When* attribute, specifying a list of conditions (i.e., its body). These conditions should be satisfied at the time a new fact is received for the corresponding subsumption to take place. To illustrate its use, we can consider another subsumption meta-fact for the *ComfortLevelMonitor* component of our running example. The following subsumption meta-fact expresses that a new `humidityReading` from a certain tent `?tent` subsumes all older humidity readings from that same tent:

```
1 subsumes!(Incoming=humidityReading(Percent = ?new)@[from(MAC=?mac,ID=?id)],
2           Fact=humidityReading(Percent = ?old)@[from(MAC=?othermac,ID=?otherid)],
3           When=[online(Tent = ?tent)@[from(MAC=?mac,ID=?id)],
4                 online(Tent = ?tent)@[from(MAC=?othermac,ID=?otherid)]]).
```

Listing 3.14: Specifying a subsumption strategy for humidityReading facts

When a new humidity reading is received and this meta-fact is processed by the Reification Engine, `?mac` and `?id` will be bound to the new reading's originating MAC address and component ID, respectively. As a result, the condition on line 3 will bind `?tent` to the originating tent and only the humidity readings from that same tent will be subsumed due to the condition on line 4.

As can be observed, the programmer has the full power of the logical reasoning engine for expressing her subsumption strategies. It must be noted, however, that the programmer cannot expect the variables used in the *Incoming* fact specification to be bound before the *When* clause is evaluated. Consequentially, these variables cannot be used where a bound variable is expected (e.g., as the argument of a method invocation). In addition, while the *When* clause might be unsatisfied after a subsumption, this operation is irreversible and will not be undone.

3.8.2 Dropping Incoming Facts

As various facts can be published to the decentralized event-bus, it is possible that a component is not interested in certain facts and hence shouldn't add these facts to its Fact Base upon reception. The drop meta-fact can be used to specify exactly those irrelevant facts. To illustrate its use, we can reconsider the running example. If for some reason, the *HeatingController*, *TemperatureSensor* and *HumiditySensor* would receive the other components' `online` facts, these facts could be dropped upon reception because they're never used by these components and hence would only pollute their Fact Bases[11]. To this end, the

[11] This would be the case if the `online` facts are published by a rule and carry the *to(MAC=any)* meta-attribute, implying a broadcast.

drop meta-fact from Listing 3.15 could be added to each of those components.

```
1 drop!(Incoming=online(Tent = ?tnt, Component = ?c)).
```

Listing 3.15: Dropping irrelevant facts

Just as in a subsumption meta-fact, the *Incoming* attribute specifies the newly received fact, which in this case has to be dropped. Whenever an incoming fact is dropped, it won't trigger any subsumption meta-facts and it won't be asserted in the Fact Base. Consequentially, it also won't trigger any interaction rules.

Additionally, a drop meta-fact can also have a *When* attribute, specifying a list of conditions (i.e., its body) to be satisfied at the time a new fact is received for the corresponding fact drop to take place. An example illustrating its use consists of storing only the maximum received temperature in the Fact Base. To obtain this behavior, the following meta-facts can be added to the CrimeSPOT runtime:

```
1 drop!(Incoming=temperatureReading(Celsius=?t),
2       When=[temperatureReading(Celsius >= ?t)]).
3
4 subsumes!(Incoming=temperatureReading(Celsius=?t),
5           Fact=temperatureReading(Celsius = ?oldT),
6           When=[?oldT < ?t]).
```

With these meta-facts in the Configuration Base, the Reification Engine will only accept a new temperature if it's strictly larger than any temperature already in the Fact Base (lines 1-2). In addition, an accepted temperature will subsume all smaller temperatures in the Fact Base (lines 4-6). Therefore, at any time, the Fact Base will contain at most one `temperatureReading`, which is the largest one received up till that moment. Again, the programmer cannot expect the variables used in the *Incoming* fact specification to be bound before the *When* clause is evaluated, and a drop operation will not be undone when the *When* clause is unsatisfied after the drop took place.

While up till now, the specifications used in the meta-facts employed variables for the attributes' values, it must be noted that constants can also be used for more specific matching. For instance, if one wants to drop all incoming sensor readings from a certain MAC address "1234:1234:1234:1234", the meta-fact from Listing 3.16 can be added to the CrimeSPOT runtime.

```
1 drop!(Incoming=sensorReading(Type=?v)@[from(MAC="1234:1234:1234:1234")]).
```

Listing 3.16: Using constants in meta-facts

3.9 Revisiting the Running Example

Having introduced the CrimeSPOT language, we will now revisit the running example and finalize its CrimeSPOT implementation. To this end, we will use the functional requirements identified earlier (cfr. Table 3.1). As most of the implementation was already explained in the previous sections, we will only discuss the remaining parts.

The sensing components For the sensing components (i.e., the *HumiditySensor* and *TemperatureSensor*), the following functional requirements were identified:

F.R.1	The *HumiditySensor* and *TemperatureSensor* have to publish their sensor readings at set intervals.
F.R.2	The *HumiditySensor* and *TemperatureSensor* have to publish their online presence and location at set intervals.

Clearly, these requirements are easily met by using interaction rules in combination with the *is*-operator to invoke the required application logic and publish the results at set intervals. The interaction rule for publishing humidity readings was already depicted in Listing 3.8. The interaction rules for publishing temperature readings and `online` facts (i.e., facts encapsulating a component's location) are very similar, which is why only the latter is shown below.

```
1 online(Tent = ?t, Component = 'TemperatureSensor)@[to(MAC=any),
2                                                    factExpires(Seconds=3600)]
3    <- ?t is getTentBasedOnGPSReading()@[renewEvery(Seconds=3600)].
```

Listing 3.17: Publishing online facts from the TemperatureSensor component using the is operator

This rule should be added to the *TemperatureSensor* component and publishes an `online` fact every hour. To this end, the identifier for the current tent is obtained by reading out a GPS sensor in the *getTentBasedOnGPSReading* method, which is invoked every hour. After substituting the value for the Component attribute in the `online` fact, this rule can also be added to the *HumiditySensor* component.

The HeatingController component For the *HeatingController* component, the following functional requirements were identified:

F.R.2	The *HeatingController* has to publish its online presence and location at set intervals.
F.R.3	The *HeatingController* has to adjust its associated heater according to a received `adjustHeating` event.
F.R.4	The *HeatingController* has to make sure that its associated heater won't keep heating when the *ComfortLevelMonitor* fails or gets disconnected from the WSN.

While F.R.2 can be met through the rule from Listing 3.17 (i.e., after substituting the value for the Component attribute in the `online` fact), F.R.3 and F.R.4 require an interaction rule that invokes application logic upon triggering. This rule can be added to the CrimeSPOT runtime as shown below.

```
1 registerRule("adjustHeating(Level = ?h)",
2              new Action() {
3                public void activated(TypedObject args) {
4                  // adjust the heater using args.getValue("h")
5                }
6                public void deactivate(TypedObject args) {
7                  // reset the heater
8                }
9              });
```

Listing 3.18: Reacting to adjustHeating facts

As soon as the *HeatingController* receives an `adjustHeating` fact, this rule will be activated and the component will steer its heater accordingly. To deal with F.R.4, an `adjustHeating` fact expires after a specific amount of time, after which this rule will be deactivated and the component will reset its heater to a neutral setting. Additionally, as we've discussed before, only the last `adjustHeating` fact should be stored in the Fact Base. To this end, the subsumption meta-fact from Listing 3.12 should also be added to the component.

The ComfortLevelMonitor component Finally, the *ComfortLevelMonitor* component had the following functional requirements:

F.R.5	The *ComfortLevelMonitor* has to compute a tent's heating level based on a received `temperatureReading` event and publish this level in an `adjustHeating` event.
F.R.6	The *ComfortLevelMonitor* has to control the heating for each tent individually by sending `adjustHeating` events only to the *HeatingControllers* in the tent to be heated.
F.R.7	The *ComfortLevelMonitor* has to relate received `humidityReading` and `temperatureReading` events that originate from the same tent and use them to compute and log that tent's comfort level.
F.R.8	The *ComfortLevelMonitor* has to make sure that only the most recent sensor readings from a certain tent are used for computing the heating- and comfort levels.

As discussed earlier, F.R.7 can be met by adding three rules to the *ComfortLevelMonitor* component. First of all, the humidity- and temperature readings have to be related to the `online` facts to obtain their originating tent. For humidity readings, this can be done through the preprocessing rule depicted in Listing 3.5, which relates these readings with `online` facts, and asserts `humidityInTent` facts. The rule for asserting `temperatureInTent` facts is very similar and therefore not shown for brevity[12]. The third rule invokes application logic to compute and log a tent's comfort level, and was depicted in Listing 3.11.

To deal with F.R.5 and F.R.6, the following interaction rule can be added to the *ComfortLevelMonitor* component:

```
1 adjustHeating(Level = ?heatingLevel)@[to(MAC=?hcm, ID=?hci),
2                                       factExpires(Seconds = 600)]
3    <- temperatureInTent(Celsius = ?temp, Tent = ?tnt),
4       ?heatingLevel is computeHeatingLevel((Number)?temp),
5       online(Component = 'HeatingController,
6              Tent = ?tnt)@[from(MAC = ?hcm, ID=?hci)].
```

As soon as a `temperatureInTent` fact is asserted (i.e., as soon as a new temperature reading is received), the condition on line 3 will be matched and will bind `?temp` to the measured temperature and `?tnt` to the originating tent. The subsequent method invocation on line 4 will compute the required heating level,

[12]Remember that these preprocessing rules require the presence of exactly one `online` fact in the Fact Base for each running component and that it's therefore also a good idea to add the subsumption meta-fact from Listing 3.13 to the CrimeSPOT runtime.

which has to be published to the *HeatingControllers* in the tent to be heated. To this end, the condition on line 5 will bind `?hcm` and `?hci` to such a *HeatingController*'s MAC address and component ID, respectively, and the `adjustHeating` fact in the rule's head is published using these values. Note that the condition on line 5 will have a match for every *HeatingController* in the tent to be heated and that this rule will consequentially send the `adjustHeating` fact to each of these controllers. As mentioned before, the `adjustHeating` fact should expire, which is expressed by its *factExpires* meta-attribute.

F.R.8 has to be dealt with using subsumption meta-facts. Because all facts in the Fact Base are taken into account for triggering the interaction rules, and consequentially for computing the heating- and comfort levels, this requirement actually states that every newly received sensor reading from a certain tent subsumes any older sensor reading from that tent. This subsumption strategy was expressed earlier for `humidityReading` facts in Listing 3.14. The subsumption strategy for `temperatureReading` facts, which is entirely analog, is shown below.

```
1 subsumes!(Incoming=temperatureReading(Celsius=?new)@[from(MAC=?mac,ID=?id)],
2           Fact=temperatureReading(Celsius=?old)@[from(MAC=?othermac,ID=?otherid)],
3           When=[online(Tent=?tnt)@[from(MAC=?mac,ID=?id)],
4                 online(Tent=?tnt)@[from(MAC=?othermac,ID=?otherid)]]).
```

Both subsumption strategies make sure that for every tent, only the last `temperatureReading` and the last `humidityReading` will be stored in the Fact Base, even when multiple redundant *TemperatureSensor*- or *HumiditySensor* components have been deployed in the same tent.

3.10 Conclusion

When programming wireless sensor network applications from a node-level perspective using event-based component middleware, every component has to be individually programmed by specifying its interactions with other components. These interactions are driven by the exchange of events over a decentralized event-bus. However, without additional support, *event handlers* are the only means to react to the reception of events.

This chapter introduced CrimeSPOT, our programming language that provides additional support for programming interactions among WSN components. Rather than using event handlers, it advocates the use of *declarative interaction rules* to react to the reception of events. To this end, CrimeSPOT reifies events as *facts* and stores them in a Fact Base. By allowing the programmer to associate an expiration time with reified events, we reconcile the transient nature of events with the persistent nature of facts. The interaction rules, each specifying one or more facts to react upon, are re-evaluated whenever the Fact Base is updated, which indirectly corresponds to the reception or expiration of events. As soon as an interaction rule's conditions are met, its reaction is triggered. In addition, a rule is also triggered when its conditions are no longer met, which allows its reaction to be compensated for.

By using CrimeSPOT, the programmer no longer has to deal with the *accidental* complexity inherent to the use of event-based WSN middleware. She no longer has to dispatch a received event to its appropriate reaction in an event handler, nor is she required to implement ad-hoc state management to keep track

of which events have already been received and whether or not they are still valid. More importantly, CrimeSPOT's interaction rules facilitate untangling the code concerned with application logic and component interactions.

More details about the implementation of the CrimeSPOT language can be found in Appendix A. The next chapter introduces the Macro-CrimeSPOT extension, which allows programming CrimeSPOT components from a network-level perspective.

⟨*expression*⟩ ::= ⟨*expr*⟩ `.`

⟨*expr*⟩ ::= ⟨*mapping*⟩
| ⟨*meta-fact*⟩
| ⟨*fact*⟩ | ⟨*rule*⟩

⟨*mapping*⟩ ::= `mapping` ⟨*fact*⟩ `<=>` ⟨*event*⟩

⟨*meta-fact*⟩ ::= ⟨*subsumption*⟩ | ⟨*drop*⟩

⟨*subsumption*⟩ ::= `subsumes!(Incoming=`⟨*fact*⟩`,Fact=`⟨*fact*⟩ [`,When=[`⟨*body*⟩`]`] `)`

⟨*drop*⟩ ::= `drop!(Incoming=`⟨*fact*⟩ [`,When=[`⟨*body*⟩`]`] `)`

⟨*fact*⟩ ::= ⟨*type*⟩`(` [⟨*attribute*⟩ (`,` ⟨*attribute*⟩)*] `)` [`@[` ⟨*meta-attributes*⟩ `]`]

⟨*attribute*⟩ ::= ⟨*name*⟩ ⟨*relop*⟩ (⟨*value*⟩ | ⟨*variable*⟩)

⟨*meta-attributes*⟩ ::= ⟨*meta-attribute*⟩ (`,` ⟨*meta-attribute*⟩)*

⟨*meta-attribute*⟩ ::= ⟨*assert-meta-attribute*⟩
| ⟨*condition-meta-attribute*⟩

⟨*event*⟩ ::= ⟨*type*⟩`(`⟨*field*⟩ (`,` ⟨*field*⟩)*`)`

⟨*field*⟩ ::= ⟨*type*⟩ `=` ⟨*variable*⟩

⟨*rule*⟩ ::= ⟨*fact*⟩ `<-` ⟨*body*⟩

⟨*body*⟩ ::= ⟨*condition*⟩ (`,` ⟨*condition*⟩)*

⟨*condition*⟩ ::= ⟨*fact*⟩
| ⟨*extralogical_condition*⟩

⟨*value*⟩ ::= ⟨*number*⟩ | ⟨*quoted_text*⟩ | ⟨*symbol*⟩ | ⟨*list*⟩

⟨*list*⟩ ::= `[` [(⟨*value*⟩ | ⟨*variable*⟩) (`,` (⟨*value*⟩ | ⟨*variable*⟩))*] `]`

⟨*quoted_text*⟩ ::= `"`⟨*text*⟩`"`

⟨*symbol*⟩ ::= `‘`⟨*text*⟩

⟨*type*⟩ ::= ⟨*text*⟩

⟨*name*⟩ ::= ⟨*text*⟩

⟨*variable*⟩ ::= `?`⟨*name*⟩

⟨*relop*⟩ ::= `=` | `!=` | `<=` | `>=` | `<` | `>`

⟨*assert-meta-attribute*⟩ ::= `to(MAC=(`⟨*quoted_text*⟩ | `any))`
| `to(ID=(`⟨*number*⟩ | `any))`
| `factExpires(Seconds=`⟨*number*⟩`)`

⟨*condition-meta-attribute*⟩ ::= `from(MAC=(`⟨*quoted_text*⟩ | ⟨*variable*⟩ | `this.MAC))`
| `from(ID=(`⟨*number*⟩ | ⟨*variable*⟩ | `this.ID))`
| `matchExpires(Seconds=`⟨*number*⟩`)`
| `matchEvery(Seconds=`⟨*number*⟩`)`

⟨*extralogical_condition*⟩ ::= (⟨*value*⟩ | ⟨*variable*⟩) `is` ⟨*methodinvocation*⟩
| `findall(`⟨*variable*⟩`,[`⟨*body*⟩`],`⟨*variable*⟩`)`
| `length(`⟨*list*⟩`,(`⟨*value*⟩ | ⟨*variable*⟩`))`
| `not` ⟨*fact*⟩
| ⟨*variable*⟩ ⟨*relop*⟩ (⟨*value*⟩ | ⟨*variable*⟩)

⟨*methodinvocation*⟩ ::= ⟨*name*⟩`(` [⟨*argument*⟩ (`,` ⟨*argument*⟩)*] `)` [`@[`⟨*option*⟩`]`]

⟨*argument*⟩ ::= ⟨*value*⟩ | `(`⟨*typecast*⟩`)`⟨*variable*⟩

⟨*typecast*⟩ ::= `Number` | `Fraction` | `Text` | `Symbol` | `List`

⟨*option*⟩ = `evalEvery(Seconds=`⟨*number*⟩`)`
| `renewEvery(Seconds=`⟨*number*⟩`)`

Figure 3.8: CrimeSPOT Grammar

4

Network-centric Programming with Macro-CrimeSPOT

So far, we have introduced the CrimeSPOT programming language which facilitates programming individual wireless sensor network components. Next to significantly reducing the complexity of dealing with events in such components, it also untangles the code concerned with application logic and component interactions. However, a few important shortcomings can be identified when programming WSN applications using CrimeSPOT.

First of all, each component is typically programmed in an individual Java class. This makes it difficult to maintain an overview on a large WSN application (i.e., an application consisting of several components). For instance, to find out which components interact with each other, a programmer carefully has to go through each individual class to inspect the interaction rules that were added to the CrimeSPOT runtime.

Second, while (nearly) identical interaction rules, event-fact mappings, and meta-facts are often required in multiple components of the same or different WSN applications, they cannot be reused through any other means than code duplication.

Inspired by the state of the art presented in Chapter 2, we decided to tackle these shortcomings by extending node-centric CrimeSPOT with a network-centric variant. Concretely, we extended CrimeSPOT with a macroprogramming language, dubbed Macro-CrimeSPOT, in which CrimeSPOT components can be programmed from a network-level perspective and in which means for code reuse are provided. Macro-CrimeSPOT comes with a precompiler, which compiles the macro-code to node-level code. When using Macro-CrimeSPOT, the application code is no longer cluttered with Java statements (e.g., to install CrimeSPOT expressions) and a lucid view on the entire WSN application is maintained. As a result, the interactions among the components become clearer and possible opportunities for code reuse become apparent.

In this chapter, we introduce Macro-CrimeSPOT and its accompanying

precompiler, and discuss the final implementation of our running example.

4.1 Grammar

Figure 4.1 gives an overview of the MACRO-CRIMESPOT language. It depicts the language's grammar using a Backus Naur Form notation extended with large square brackets to depict optional items and large parenthesis suffixed with an asterisk to depict zero or more repetitions. Note that <*cs_expression*> refers to

⟨*application*⟩	::= (⟨*component_block*⟩)*
⟨*component_block*⟩	::= ⟨*quantifier*⟩ ⟨*code_block*⟩
⟨*quantifier*⟩	::= `*` | ⟨*name*⟩ (`,` ⟨*name*⟩)*
⟨*code_block*⟩	::= ⟨*crimespot_block*⟩ | ⟨*java_block*⟩
⟨*crimespot_block*⟩	::= `{` ⟨*crimespot_code*⟩ `}`
⟨*java_block*⟩	::= `:java {` ⟨*java_code*⟩ `}`
⟨*crimespot_code*⟩	::= (⟨*cs_expression*⟩`.` | ⟨*mcs_expression*⟩)*
⟨*mcs_expression*⟩	::= `defvar` ⟨*meta-var*⟩`:` ⟨*meta-var-value*⟩`.` | `defmacro` ⟨*name*⟩`(` [⟨*meta-var*⟩ (`,` ⟨*meta-var*⟩)*] `):` ⟨*cs_expression*⟩`.` | ⟨*macro*⟩`.` | `import!("`⟨*library_file_path*⟩`").`
⟨*meta-var*⟩	::= `$`⟨*name*⟩
⟨*macro*⟩	::= ⟨*name*⟩`(` [⟨*meta-var-value*⟩ (`,` ⟨*meta-var-value*⟩)*] `)`
⟨*library_file_path*⟩	::= ⟨*text*⟩
⟨*library*⟩	::= ⟨*crimespot_code*⟩
⟨*name*⟩	::= ⟨*text*⟩

Figure 4.1: Macro-CrimeSPOT Grammar

<*expr*> from the CRIMESPOT grammar, which was depicted in Figure 3.8. A MACRO-CRIMESPOT application can be written in a text file and compiled by the precompiler, which generates the node-level Java code for the application's components. In the following sections, we will introduce all language features shown in the grammar.

4.2 Grouping the Components of a WSN Application

The code base of a CRIMESPOT WSN application is typically small. Each component is written in a Java class and consists of only a few event-fact mappings, meta-facts, interaction rules and Java methods. Hence, a straightforward way to switch to the network-level perspective is by collecting the code concerning a single application in one source file.

In MACRO-CRIMESPOT, every application is represented as a single file containing *blocks* of code for each component. There are two kinds of blocks: blocks that group the CRIMESPOT expressions governing a component's interactions, and blocks that group the Java code implementing a component's

application logic. To introduce the syntax, Listing 4.1 depicts an example application consisting of two components.

```
1 TemperatureSensor {
2   online(Tent = ?t, Component = 'TemperatureSensor)@[to(MAC=any),
3                                                      factExpires(Seconds=3600)]
4       <- ?t is getTentBasedOnGPSReading()@[renewEvery(Seconds=3600)].
5
6   temperatureReading(Celsius = ?temp)@[to(MAC=any),
7                                        factExpires(Seconds = 600)]
8       <- ?temp is getTemperature()@[renewEvery(Seconds = 600)].
9 }
10
11 HumiditySensor {
12   // publish online and humidityReading facts
13 }
```

Listing 4.1: Macro-CrimeSPOT syntax

Note that each block is delimited by curly braces and preceded by a *quantifier*. This quantifier indicates the component to which the code from within the corresponding block should be added. In this example, the blocks group the CrimeSPOT expressions for the *TemperatureSensor*- and *HumiditySensor* components from our running example.

As it's often required to introduce rules that invoke application logic, and to specify their corresponding application logic in Java, every quantifier can also take the `:java` suffix to indicate that the corresponding block's content is plain Java code. To illustrate this, we can reconsider the *HeatingController* component from our running example. This component reacts to `adjustHeating` facts and steers its associated heater accordingly. Its partial macro-implementation is given below.

```
1 HeatingController {
2   // ...
3   this.adjustHeater
4     <- adjustHeating(Level = ?h).
5 }
6 HeatingController:java {
7   private Action adjustHeater = new Action() {
8     public void activated(TypedObject args) { ... }
9     public void deactivate(TypedObject args) { ... }
10  };
11 }
```

Listing 4.2: Implementing the HeatingController in Macro-CrimeSPOT

The precompiler will add all Java code to the component's Java class without further processing. Note that the Java variable bound to the instance of the abstract *Action* type is accessible from within the interaction rule's head by prefixing it with `this.`

4.2.1 Quantifying over Multiple Components

When considering the code from Listing 4.1, opportunities for code reuse already become apparent. Since both the *TemperatureSensor* and *HumiditySensor* components publish their presence using an `online` fact, the corresponding interaction rule could just be added to each component by the precompiler rather than manually duplicating it. To this end, Macro-CrimeSPOT allows the

programmer to quantify code blocks over multiple components. Such quantification can be achieved by either listing multiple component names as a block's quantifier, or by using the universal * quantifier.

The use of the universal * quantifier is illustrated in Listing 4.3. When this quantifier is used, the precompiler will add all code specified within the corresponding block to each component in the application.

```
1 * {
2   online(Tent = ?t, Component = '$COMPONENT_NAME)@[to(MAC=any),
3                                                    factExpires(Seconds=3600)]
4      <- ?t is getTentBasedOnGPSReading()@[renewEvery(Seconds=3600)].
5 }
6
7 *:java {
8     private Attribute getTentBasedOnGPSReading() { ... }
9 }
10
11 TemperatureSensor { /* publish temperature readings */ }
12 HumiditySensor { /* publish humidity readings */ }
```

Listing 4.3: Illustrating the universal quantifier

For this example, the same behavior could be obtained by using `TemperatureSensor, HumiditySensor` as the blocks' quantifier. Listing destination components is especially useful in applications with multiple components where specific code should be added to some, but not all, components.

4.2.2 Using Meta-Variables in CrimeSPOT Expressions

When developing an application using Macro-CrimeSPOT, the programmer can employ *meta-variables* in her CrimeSPOT expressions. Unlike CrimeSPOT's variables, Macro-CrimeSPOT's meta-variables are only valid at compile-time and will be substituted by their value by the precompiler. Meta-variables are prefixed by a `$` sign and are either predefined by the precompiler, or defined by the programmer in the scope of a particular component.

Within a CrimeSPOT code block, the programmer can use the predefined `$COMPONENT_NAME` meta-variable wherever the component's name is expected. This is often useful when quantifying over multiple components. For instance, in Listing 4.3, the `online` fact published by a component has to contain the component's name, which is therefore represented by `$COMPONENT_NAME`.

Other meta-variables can also be introduced by the programmer. This is, for instance, useful when dealing with several sensing components which publish facts at set intervals. To configure such an interval for the entire application, the `defvar` statement can be used within a universally quantified CrimeSPOT block to bind the interval to a meta-variable.

```
1 * {
2    defvar $publicationInterval: Seconds = 600.
3 }
```

When using the above code in combination with the *TemperatureSensor* and *HumiditySensor* components from Listing 4.1, the explicit intervals in these components' code can be replaced by the `$publicationInterval` meta-variable. As such, the application becomes easier to reconfigure.

In general, *meta-variables* and their definitions can only be used within CrimeSPOT code blocks. The meta-variables can be bound to any textual

expression without dots, in which commas appear only within balanced parentheses or brackets. Typically, meta-variables are used for representing very small snippets of CrimeSPOT code.

4.2.3 Abstracting CrimeSPOT Expressions

While the possibility to quantify code blocks over multiple components allows to implicitly reuse code in certain components, any full-fledged CrimeSPOT expression can also be reused by abstracting it as a *macro* and explicitly referring to it from within a CrimeSPOT code block.

As an example, we can consider a macro for reusing an event-fact mapping for temperature readings. If we assume that the `$publicationInterval` meta-variable is bound where the macro will be used[1], the macro-definition from Listing 4.4 can be used for this purpose.

```
1 * {
2    defmacro temperatureMapping():
3       mapping temperatureReading(Celsius=?temp)@[factExpires($publicationInterval)]
4          <=> Event_101(Integer=?temp).
5 }
```

Listing 4.4: Macro for the temperature event-fact mapping

Because the macro is defined in a universally quantified code block, its definition is available to all components. They can refer to the macro as follows:

```
1 temperatureMapping().
```

Whenever the precompiler encounters a macro-name within a CrimeSPOT code block, it will expand the macro as specified in its definition. Since in our running example, only the *TemperatureSensor* and *ComfortLevelMonitor* components employ temperature readings, this macro can be used in exactly those components.

If we further reconsider our running example, another useful macro would represent the interaction rule for publishing the `online` facts. Since only the *TemperatureSensor*, *HumiditySensor* and *HeatingController* components should publish such facts, this macro could be used in exactly those components. A possible macro-definition is given in Listing 4.5. This macro takes the `$time` meta-variable as an argument.

```
1 * {
2     defmacro publishPresenceEvery($time):
3         online(Tent = ?tnt, Component = '$COMPONENT_NAME)@[to(MAC=any),
4                                                            factExpires($time)]
5            <- ?tnt is getTentBasedOnGPSReading()@[renewEvery($time)].
6 }
```

Listing 4.5: Macro for publishing a component's presence

In general, a macro can take any number of meta-variables as arguments for which the values have to be passed whenever the macro is used. As such, expanding a macro also binds meta-variables, but only during this expansion. For instance, the code below can be added to any component to publish an `online` fact every hour.

```
1 publishPresenceEvery(Seconds=3600).
```

[1] Our macros are dynamically scoped.

4.2.4 Organizing Abstract CrimeSPOT Expressions into Libraries

Finally, we also allow CrimeSPOT expressions and macro- and meta-variable definitions to be stored in external files or *libraries*. Such a library can be imported from within a CrimeSPOT code block using the `import!` statement as shown below.

```
1 import!("/path/to/library.mcs").
```

The precompiler will substitute this statement by the imported library's contents. By providing this feature, our macroprogramming language allows to easily reuse code among various WSN applications (e.g., a library with general macro-definitions or a full-fledged component extension).

4.3 Compiling Macro-Code

As mentioned before, Macro-CrimeSPOT comes with a precompiler that compiles the macro-code to node-level Java code for each component. Our implementation's source code can be found online[2]. While our precompiler targets CrimeSPOT on the SunSPOT platform with the LooCI event-based component middleware, it could easily be adapted to target any other platform. Figure 4.2 gives an overview of how the precompiler works.

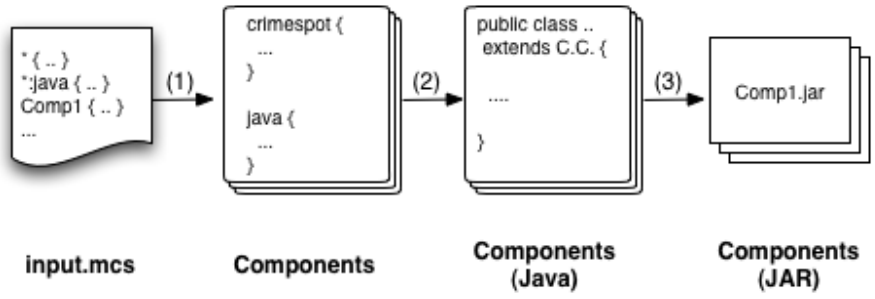


Figure 4.2: Macro-CrimeSPOT Precompiler

As can be observed, compiling macro-code is quite straightforward and typically a three phase process.

In the *first phase*, the Macro-CrimeSPOT source is parsed to extract the individual components' CrimeSPOT- and Java code blocks. In case certain code blocks were quantified over multiple components, the contents of these blocks will be added to the corresponding blocks of the individual components.

In the *second phase*, every component is individually processed. Its CrimeSPOT code block is parsed to detect errors in the expressions[3] and to do macro-expansions, imports, and meta-variable substitutions. In addition, all methods invoked from within the component's interaction rules are detected and the dispatcher method (cfr. Section A.4) is dynamically generated. To end this

[2] `http://soft.vub.ac.be/amop/crime/sunspot`

[3] Since all CrimeSPOT expressions are installed at runtime, this error detection makes sure that all possible errors are detected at compile-time.

phase, the required Java statements to install the CrimeSPOT expressions are generated and all Java code is merged in a full-fledged Sun SPOT Application for the component.

The *third phase* is optional. In case the components' node-level Java code doesn't have to be refined, the precompiler can produce a JAR for every component, which is ready for deployment on WSN nodes using LooCI's over-the-air deployment tools.

4.4 Revisiting the Running Example

Having introduced Macro-CrimeSPOT, we will now illustrate its use by revisiting the running example from a network-perspective. The running example was explained in Section 3.1 and concerns a wireless sensor network application for controlling the heating and logging the comfort levels of festival tents. Since all Java code has already been shown, we won't discuss the Java code blocks for brevity.

Listing 4.6 shows the application's **universally quantified code block**, which provides the code to be added to each application component.

```
1 * {
2     import!("mappings-library.mcs").
3
4     defvar $readingPublicationInterval: Seconds = 600.
5     defvar $onlinePublicationInterval: Seconds = 3600.
6
7     defmacro publishPresenceEvery($time):
8        online(Tent = ?tnt, Component = '$COMPONENT_NAME)@[to(MAC=any),
9                                                          factExpires($time)]
10          <- ?tnt is getTentBasedOnGPSReading()@[renewEvery($time)].
11
12    defmacro newReadingSubsumesOlderFromSameTent($reading, $type):
13       subsumes!(Incoming= $reading($type = ?new)@[from(MAC=?mac)],
14                 Fact=$reading($type = ?old)@[from(MAC=?othermac)],
15                 When=[online(Tent = ?tnt)@[from(MAC=?mac)],
16                       online(Tent = ?tnt)@[from(MAC=?othermac)]]).
17 }
```

Listing 4.6: Running Example - Universally quantified code block

Since event-fact mappings are used in various WSN applications, the default ones have been put in a library that is imported in this application (line 2). As shown in Listing 4.7, this library contains two interesting macros, namely `temperatureMapping($factExp)` and `humidityMapping($factExp)`, which provide exactly the event-fact mappings required for this application.

```
1 defmacro temperatureMapping($factExp):
2     mapping temperatureReading(Celsius=?temp)@[factExpires($factExp)]
3         <=> Event_101(Integer=?temp).
4
5 defmacro humidityMapping($factExp):
6     mapping humidityReading(Percent=?h)@[factExpires($factExp)]
7         <=> Event_102(Integer=?h).
8
9 //...
```

Listing 4.7: Running Example - Event-Fact Mappings Library

Next to importing the library, two meta-variables and two application-specific macros are defined. While the meta-variables configure the time intervals relevant to this application, the macros define some reusable code. For instance, the `newReadingSubsumesOlderFromSameTent($reading, $type)` macro can be used to install a subsumption meta-fact which specifies that a new `$reading` fact from a certain tent subsumes all older `$reading` facts from the same tent.

The implementation for the **sensing components** and the **HeatingController component** is shown in Listing 4.8. Compared to the implementation given earlier, the required event-fact mappings are now installed using the library macros (lines 19, 28) and the interaction rule for publishing the `online` fact is installed using the `publishPresenceEvery` macro (lines 20, 29, 37).

```
18 TemperatureSensor {
19     temperatureMapping($readingPublicationInterval).
20     publishPresenceEvery($onlinePublicationInterval).
21
22     temperatureReading(Celsius = ?temp)@[to(MAC=any),
23                                          factExpires($readingPublicationInterval)]
24       <- ?temp is getTemperature()@[renewEvery($readingPublicationInterval)].
25 }
26
27 HumiditySensor {
28     humidityMapping($readingPublicationInterval).
29     publishPresenceEvery($onlinePublicationInterval).
30
31     humidityReading(Percent = ?p)@[to(MAC=any),
32                                    factExpires($readingPublicationInterval)]
33       <- ?p is getHumidity()@[renewEvery($readingPublicationInterval)].
34 }
35
36 HeatingController {
37     publishPresenceEvery($onlinePublicationInterval).
38
39     subsumes!(Incoming=adjustHeating(Level = ?new),
40               Fact=adjustHeating(Level = ?old)).
41
42     this.adjustHeater
43       <- adjustHeating(Level = ?h).
44 }
```

Listing 4.8: Running Example - Sensor- and HeatingController components

In addition, the interaction rules for publishing the sensed readings now employ the `$readingPublicationInterval` meta-variable to specify the interval at which the readings should be published. Because a published sensor reading fact expires just before a new sensor measurement is taken, no retraction events have to be published on the underlying event-bus when these interaction rules are triggered[4].

```
45 ComfortLevelMonitor {
46     temperatureMapping($readingPublicationInterval).
47     humidityMapping($readingPublicationInterval).
48
49     subsumes!(Incoming=online(Tent = ?tnt, Component = ?c)@[from(MAC=?m)],
50               Fact=online(Tent = ?otnt, Component = ?c)@[from(MAC=?m)]).
51
```

[4]In case the published fact had not expired at the time a new match for the *is*-operator is obtained and the old one consequentially becomes invalid, a retraction for the fact would have to be published because the interaction rule got *deactivated* for the invalidated match.

```
52      newReadingSubsumesOlderFromSameTent(humidityReading, Percent).
53      newReadingSubsumesOlderFromSameTent(temperatureReading, Celsius).
54
55      this.logComfortLevel
56        <- humidityReading(Percent = ?h)@[from(MAC=?hm, ID=?hi)],
57           temperatureReading(Celsius = ?t)@[from(MAC=?tm, ID=?ti)],
58           online(Tent = ?tnt)@[from(MAC = ?hm, ID = ?hi)],
59           online(Tent = ?tnt)@[from(MAC = ?tm, ID = ?ti)].
60
61      adjustHeating(Level = ?heatingLevel)@[to(MAC=?hcm, ID=?hci),
62                                            factExpires($readingPublicationInterval)]
63        <- temperatureReading(Celsius = ?t)@[from(MAC=?tm, ID=?ti)],
64           online(Tent = ?tnt)@[from(MAC = ?tm, ID = ?ti)],
65           ?heatingLevel is computeHeatingLevel((Number)?t),
66           online(Component = 'HeatingController,
67                  Tent = ?tnt)@[from(MAC = ?hcm, ID=?hci)].
68 }
```

Listing 4.9: Running Example - ComfortLevelMonitor component

Listing 4.9 shows the implementation for the **ComfortLevelMonitor component** and concludes the macro-implementation of our running example. Compared to the implementation given earlier, the *ComfortLevelMonitor* component now uses the application's configuration meta-variables, the mapping library macros, and the `newReadingSubsumesOlderFromSameTent` macro to reuse as much code as possible. In addition, the preprocessing rules for asserting `temperatureInTent` and `humidityInTent` facts are no longer used. As can be observed, the CrimeSPOT expressions for the entire WSN application roughly consume only 68 source lines of code (i.e., excluding the mappings library).

4.5 Conclusion

This chapter introduced Macro-CrimeSPOT, our macroprogramming language for programming WSN applications using CrimeSPOT from a network-level perspective. For shifting to a network-level perspective, Macro-CrimeSPOT merges all components' code in a single source file per application. However, as WSN components are typically small, this merging doesn't result in huge source files. On the contrary, it results in clearer code in which a lucid view on the entire WSN application is retained.

Next to the ability of programming an entire WSN application from a network-level perspective, Macro-CrimeSPOT also provides the programmer with means to reuse code and thereby avoid code duplication. While *component quantification*, *meta-variables* and *macros* can be used to reuse code within a single WSN application, code can also be imported from *libraries* and thereby reused over various applications.

The next chapter will validate both CrimeSPOT and Macro-CrimeSPOT through some additional illustrative examples.

5 Validation

To validate both CrimeSPOT and Macro-CrimeSPOT, this chapter discusses the implementation of illustrative example applications. Afterwards, we summarize the discussed applications' statistics to evaluate the expressiveness of our language.

5.1 Illustrative Examples

Wireless sensor networks are used for a plethora of applications. Some examples are monitoring applications (e.g., monitoring habitats [5], zebras [6] and glaciers [7]), emergency detection applications (e.g., detecting intrusions [8], forest fires [9] and flooding rivers [10]), and more active applications like controlling heating- and air conditioning systems [11]. In this section, we will discuss some of these applications together with their Macro-CrimeSPOT implementation to demonstrate the expressiveness of our solution.

5.1.1 Fire Detection

As illustrated in Figure 5.1, an application for detecting and fighting fire requires two types of WSN nodes: *smoke detectors* (D), which have a smoke sensor and publish smoke detection events, and *controllers* (C), which each monitor a certain region and turn on their associated sprinklers, alarms and emergency exit lights in case a fire is detected. The arrows in the Figure illustrate the node interactions in a particular region. After a controller informs its smoke detectors that they're within its region, the smoke detectors will publish their events to that controller.

5.1.1.1 Implementation

For each type of WSN node, a component can be developed and afterwards deployed on the corresponding nodes. As shown below, we start the development

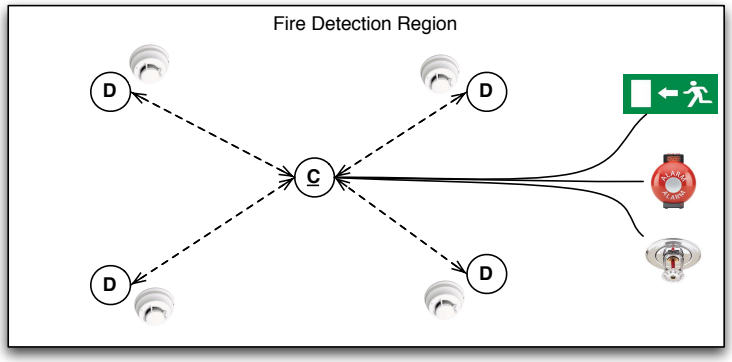


Figure 5.1: A Wireless Sensor Network application for detecting and fighting fire

by defining some meta-variables in the **universally quantified code block** to configure the application.

```
* {
    defvar $detectionInterval: Seconds=60.
    defvar $alarmThreshold: 2.
}
```

While the `$detectionInterval` configures the interval between every two subsequent smoke sensor readouts, the `$alarmThreshold` configures the amount of sensors that have to detect smoke in a region before the controller will sound the alarm.

A **Controller component** participates in two interactions. After informing its region's smoke detectors, it listens for their `smoke` facts to detect fire. Its implementation is given below.

```
Controller {
    mySensor(MAC="MAC1").
    mySensor(MAC="MAC2").
    mySensor(MAC="MACn").

    itsController()@[to(MAC=?sensorMAC)]
      <- mySensor(MAC=?sensorMAC).

    this.respond
      <- findall(?m,
                 [smoke()@[from(MAC=?m)]],
                 ?alarmingSensors),
         length(?alarmingSensors, ?l),
         ?l > $alarmThreshold.
}

Controller:java {
    private Action respond = new Action() {
        public void activated(TypedObject arguments) {
           // turn on the sprinklers, alarms, and emergency exit lights
        }
        public void deactivate(TypedObject arguments) {
          // turn off the sprinklers, alarms, and emergency exit lights
        }
    };
}
```

Note that the `mySensor` facts are statically added to the Controller's Fact Base to configure its region (lines 6-8)[1]. The presence of these facts will activate the first interaction rule, which will configure the listed smoke detectors to send their facts to the controller (lines 10-12). To this end, the Controller sends them an `itsController` fact which includes its MAC address[2].

The second interaction rule is used for detecting and reacting to fire (lines 13-18). It explicitly states when the Controller should respond; i.e., whenever more than `$alarmThreshold smoke` facts have been received and are still valid. The response itself is defined in the implementation for the `respond` *Action* instance (lines 22-29). Whenever the rule is activated, the controller will turn on its sprinklers, alarms, and emergency exit lights, and conversely, whenever the rule gets deactivated, it will turn them off.

The remaining **Smoke Detector component** can be defined using the single interaction rule shown below.

```
31 SmokeDetector {
32     smoke()@[to(MAC=?mac), factExpires($detectionInterval)]
33       <- itsController()@[from(MAC=?mac)],
34          'true is smokeDetected()@[renewEvery($detectionInterval)].
35 }
36
37 SmokeDetector:java {
38     private Attribute smokeDetected() { /* read out smoke sensor */ }
39 }
```

As soon as an `itsController` fact has been received, the smoke detector is configured and will query its sensor at intervals specified by the `$detectionInterval` meta-variable (line 34). Upon smoke detection, it will send a `smoke` fact to its controller. Clearly, a smoke detector can be configured for several controllers, in which case its Fact Base will contain several `itsController` facts. Consequentially, whenever smoke is detected, the interaction rule will be activated for every `itsController` fact and the smoke detector will send a `smoke` fact to each controller.

It must be noted that the Smoke Detector can also be implemented as a non-CrimeSPOT, plain event-based component, which might be useful when certain smoke detector sensor nodes are less powerful or unable to use CrimeSPOT because it's not available on their platform. To this end, the event-based middleware should be manually configured such that the plain smoke detectors publish their events to the correct controller, and the Controller's implementation should be augmented with an *event-fact mapping* to map these events to `smoke` facts that can be asserted in its Fact Base. If the smoke events are of type 100 and are published at the same intervals as specified by the `$detectionInterval` meta-variable, the following event-fact mapping can be used.

```
1 mapping smoke()@[factExpires($detectionInterval)]
2    <=> Event_100().
```

[1] In case several controller components are deployed within the application, every controller's region has to be individually configured. This can, for instance, be done by refining the controller's source code that was produced by the Macro-CrimeSPOT precompiler.

[2] As explained earlier, a fact's originating MAC address is implicitly present in its meta-attributes.

5.1.2 Monitoring Temperatures

Another interesting WSN application monitors the temperature in several regions, computes the average temperature in the regions and notifies the regions with the highest temperature. A possible configuration for this application is depicted in Figure5.2. The application uses only two components: the *Temperature Sensor* component (T) has been deployed in all regions and blinks a led when its region has the highest temperature, and the *Average Computer* component (C) is deployed only once to steer the application. In addition, the Average Computer is connected to a display that outputs the current average temperature. While every Temperature Sensor broadcasts its measured temperature, the Average Computer sends a notification to the Temperature Sensors in the regions with the highest temperature.

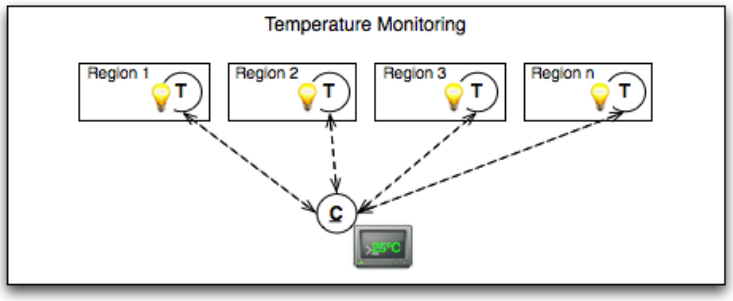


Figure 5.2: A Wireless Sensor Network application for monitoring temperatures

5.1.2.1 Implementation

The **Temperature Sensor component** has a simple implementation, which is shown below.

```
1 TemperatureSensor {
2     defvar $readingPublicationInterval: Seconds=60.
3
4     temperatureReading(Celsius = ?temp)@[to(MAC=any),
5                                          factExpires($readingPublicationInterval)]
6       <- ?temp is getTemperature()@[renewEvery($readingPublicationInterval)].
7
8     this.alertMaxTemp
9       <- maximumTemperature().
10 }
11
12 TemperatureSensor:java {
13     private Action alertMaxTemp = new Action() {
14        public void activated(TypedObject args) { /* blink led */ }
15        public void deactivate(TypedObject args) { /* turn off led */ }
16     };
17 }
```

While the first interaction rule publishes `temperatureReading` facts at intervals specified by the `$readingPublicationInterval` meta-variable, the second interaction rule reacts to the reception of a `maximumTemperature` fact, which

indicates that the component is deployed in a region that currently has the highest measured temperature and should consequentially blink its led.

Even though the implementation for the **Average Computer component** is somewhat larger, it's not necessarily more complex. The implementation is shown below. Next to two interaction rules, the component also requires some additional Java methods to do its job. However, since these Java methods are straightforward to implement, only a description of their behavior is given here.

```
18 AvgComputer {
19     this.outputAverageTemp
20       <- findall(?t, [temperatureReading(Celsius = ?t)], ?temps),
21          ?avgTemp is computeAverage((List)?temps).
22
23     maximumTemperature(Celsius = ?maxTemp)@[to(MAC=?m)]
24       <- findall(?t, [temperatureReading(Celsius = ?t)], ?temps),
25          ?maxTemp is computeMaximum((List)?temps),
26          temperatureReading(Celsius=?maxTemp)@[from(MAC=?m)].
27 }
28
29 AvgComputer:java {
30     private Action outputAverageTemp = new Action() {
31        public void activated(TypedObject args) {
32          // show args.getValue("avgTemp") on the display
33        }
34        public void deactivate(TypedObject args) { /* erase display */ }
35     };
36
37     private Attribute computeAverage(List nrLst) {
38       // compute the average Number in the List
39     }
40
41     private Attribute computeMaximum(List nrLst) {
42       // compute the maximum Number in the List
43     }
44 }
```

The first interaction rule computes the average temperature based on the current valid readings in the Fact Base. After collecting all temperatures in a List using the findall primitive, the average is computed and shown on the display.

The second interaction rule, on the other hand, is used for interacting with the Temperature Sensor components. Again, all temperature readings are collected in a List, but this time, the highest temperature is computed and bound to `?maxTemp`. Finally, the last condition of the interaction rule (line 26) uses `?maxTemp` to obtain the MAC addresses of those Temperature Sensor components that are deployed in a region that currently has the highest temperature. This interaction rule will be activated for every such component and will consequentially notify exactly those Temperature Sensors by sending them a `maximumTemperature` fact.

5.1.3 River Monitoring

In [10], Hughes et al use the LooCI event-based component middleware to deploy a river monitoring WSN application, which we have used as an inspiration for this example. Even though this application doesn't introduce a lot of new techniques, it demonstrates a WSN component that publishes sensor readings, but in addition also *autonomously* acts based on its sensed readings. A possible configuration for this application is shown in Figure5.3. Two components have

been deployed: the *River Monitor* (M), which publishes the river's water level, and the *Logger* (L), which sends its received data to a connected terminal. Next

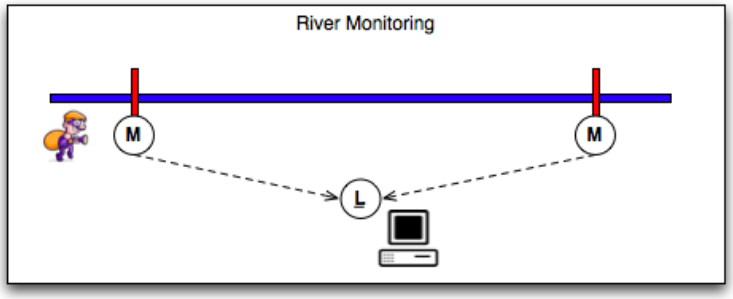


Figure 5.3: A Wireless Sensor Network application for monitoring a river

to monitoring the river's water level, the river monitor component also controls its associated sluice and detects possible theft attempts.

5.1.3.1 Implementation

Because the **Logger component** only sends its received data to a terminal by using interaction rules that invoke application logic, we will not further discuss its implementation for brevity.

The implementation of the **River Monitor component**, which is the most interesting one, is shown below. The component is configured with meta-variables that are used in its three interaction rules.

```
1 Logger { ... }
2 Logger:java { ...}
3
4 RiverMonitor {
5     defvar $logger: MAC="logger-mac".
6     defvar $verifyRiverLevelInterval: Seconds=600.
7     defvar $floodThreshold: 20.
8     defvar $verifyTheftInterval: Seconds=60.
9     defvar $theftThreshold: 1.
10
11    riverLevel(L=?l)@[to($logger),
12                      factExpires($verifyRiverLevelInterval)]
13      <- ?l is getRiverLevel()@[renewEvery($verifyRiverLevelInterval)].
14
15    possibleTheft()@[to($logger)]
16      <- ?a is getAcceleration()@[renewEvery($verifyTheftInterval)],
17         ?a > $theftThreshold.
18
19    this.controlSluice
20      <- ?l is getRiverLevel()@[renewEvery($verifyRiverLevelInterval)],
21         ?l > $floodThreshold.
22 }
```

As can be observed, a river monitor only interacts with its logger through the first two interaction rules, which are used for sending the measured river levels and possible theft detections, respectively. To detect a possible theft, the WSN node's accelerometer is read every minute to verify whether its value exceeds

the `$theftThreshold`. When a possible sensor-theft is detected, the terminal connected to the logger can, for instance, sound the alarm.

The last interaction rule is used for controlling the sluice when the measured river level rises above the predefined `$floodThreshold` (e.g., by lifting the sluice), and when it falls below it afterwards (e.g., by lowering the sluice again). Clearly, the river monitor component acts autonomously, as it never receives any facts to react upon.

5.1.4 Range Coverage

Finally, we will discuss the implementation of a Range Coverage detection system [28]. This system can be used to extend a WSN application's components and allow them to individually detect whether their range is covered by other components. A WSN configuration for the bare detection system is shown in Figure5.4. The *Range* component (R) is deployed on every WSN node and every

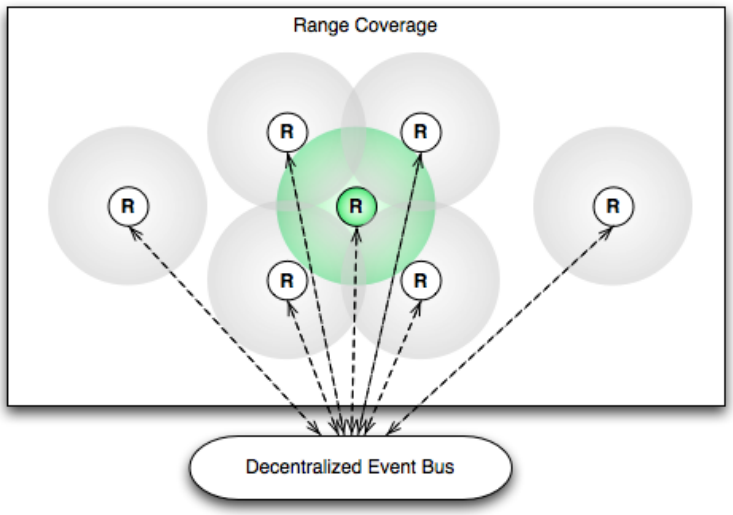


Figure 5.4: A Wireless Sensor Network application with range coverage detection

node interacts with every other node by periodically broadcasting its range over the underlying, decentralized event-bus. This allows every node to individually detect whether its range is covered by other nodes. For instance, in the knowledge of the other nodes' ranges from Figure5.4, the green node can detect that its range is covered by its four adjacent nodes.

5.1.4.1 Implementation

The first part of the **Range component**'s implementation is straightforward and shown below.

```
1 RangeComponent {
2     defvar $rangeBroadcastInterval: Seconds=3600.
3
4     subsumes!(Incoming=sensorRange(X = ?x, Y = ?y, R = ?r)@[from(MAC=?m)],
5               Fact=sensorRange(X = ?ox, Y = ?oy, R = ?or)@[from(MAC=?m)]).
```

```
6
7     sensorRange(X = ?x, Y = ?y, R = ?r)@[to(MAC=any),
8                                          factExpires($rangeBroadcastInterval)]
9       <- ?x is getX()@[renewEvery($rangeBroadcastInterval)],
10         ?y is getY(),
11         ?r is getR().
```

The component broadcasts `sensorRange` facts at intervals specified by the `$rangeBroadcastInterval` meta-variable. To this end, it obtains its range through the *getX()*, *getY()* and *getR()* methods, whose implementations are outside the scope of this example. One possible implementation could read out a GPS sensor to obtain these values.

Next to the interaction rule for broadcasting the sensor range, the component also installed a subsumption meta-fact to make sure that only the last `sensorRange` fact from a certain sensor node is kept in its Fact Base. This is important because multiple such facts could be received from the same sensor node when multiple range components have been deployed on it.

The remaining part of the implementation is concerned with detecting range coverage based on the `sensorRange` facts in the Fact Base. It is shown below.

```
12    defmacro verifyPartialCoverage($verificationMethod, $factToAssert):
13      $factToAssert(byMAC=?m)
14          <- sensorRange(X = ?x, Y = ?y, R = ?r)@[from(MAC=?m)],
15             ?myMAC is getMAC(), ?m != ?myMAC,
16             `true is $verificationMethod((Number)?x, (Number)?y, (Number)?r).
17
18    verifyPartialCoverage(coversTopLeft,topLeftIsCovered).
19    verifyPartialCoverage(coversTopRight,topRightIsCovered).
20    verifyPartialCoverage(coversBottomLeft,bottomLeftIsCovered).
21    verifyPartialCoverage(coversBottomRight,bottomRightIsCovered).
22
23    myRangeIsCovered(TL=?tlm, TR=?trm, BL=?blm, BR=?brm)
24        <- topLeftIsCovered(byMAC=?tlm),
25           topRightIsCovered(byMAC=?trm),
26           bottomLeftIsCovered(byMAC=?blm),
27           bottomRightIsCovered(byMAC=?brm).
28 }
29
30 RangeComponent:java {
31    private Attribute coversTopLeft(Number x, Number y, Number r) {
32        int myTLMinX = getXCoordinate() - getRadius();
33        int myTLMinY = getYCoordinate();
34        int myTLMaxX = getXCoordinate();
35        int myTLMaxY = getYCoordinate() + getRadius();
36
37        if(covers(x.getValue(), y.getValue(), r.getValue(),
38           myTLMinX, myTLMinY, myTLMaxX, myTLMaxY))
39            return ExpressionFactory.symbol("true");
40        else
41            return ExpressionFactory.symbol("false");
42    }
43    private boolean covers(int x, int y, int r,
44                           int bbMinX, int bbMinY,
45                           int bbMaxX, int bbMaxY) {
46        int obbMinX = x - r;
47        int obbMinY = y - r;
48        int obbMaxX = x + r;
49        int obbMaxY = y + r;
50
51        return obbMinX <= bbMinX && obbMinY <= bbMinY
```

```
52              && obbMaxX >= bbMaxX && obbMaxY >= bbMaxY;
53      }
54      ...
55 }
```

As expressed in the rule on lines 23-27, a component's range coverage is detected by splitting its range in four parts (i.e., top- and bottom left and right) and verifying whether each of these parts is covered by any other component. For every possible covering of a component's range, a `myRangeIsCovered` fact is asserted in the Fact Base, which can be reacted to by an application-specific rule. For instance, in case a component's range is covered, it can choose drop certain facts to lower its processing load.

Verifying the coverage of each partial range is done by using the `verifyPartialCoverage` macro on lines 18-21. This macro expands to a rule for verifying a specific partial range's coverage. For instance, the first macro usage (line 18) expands to a rule for verifying whether the top left part of the component's range is covered. To verify this coverage, the rule will invoke the *coversTopLeft(Number,Number,Number)* method with the data of each `sensorRange()@[from(MAC=?m)]` fact, and will assert a `topLeftIsCovered(byMAC=?m)` fact for every sensor node covering this component's top left partial range. Conversely, these facts are retracted when the partial range is no longer covered.

The Java methods for verifying the top left partial range coverage are included on lines 31-53. The implementation is straightforward: for both the component's top left partial range and the range to verify, a bounding box is created, and the method checks whether the component's bounding box is included in the other bounding box. If this is the case, the top left partial range is covered.

To extend another WSN application with this range coverage system, the code from lines 3 to 27 should be stored in a *library* and imported in the components to be extended as illustrated below. Because the range component's Java code cannot be part of the library, it should be added manually to the extended components' Java code blocks. However, this allows to specify an application-specific implementation for the required Java methods.

```
1 AnyComponent {
2     defvar $rangeBroadcastInterval: Seconds=n.
3     import!("/path/to/rangecoverage.mcs").
4
5     // other logic
6 }
7 AnyComponent:java {
8     // application-specific range coverage logic
9
10    // other logic
11 }
```

After successfully importing the range coverage system, the extended components can now use the `myRangeIsCovered` fact from their Fact Base, knowing that it will be present whenever other components cover their range.

5.2 Expressiveness

Having explained several illustrative example applications, we will now discuss the expressiveness of the CrimeSPOT language. Even though it's difficult to measure expressiveness, we made an attempt by analyzing the statistics of our example applications. These statistics are presented in Tables 5.1 and 5.2.

	Components	**Macros**	**Meta-Vars**
Running Example	4	4	2
Fire Detection	2	0	2
Temperature Monitoring	2	0	1
River Monitoring	2	0	5
Range Coverage	1	1	1
Average/Application	2,2	1	2,2

Table 5.1: Example Applications: General Statistics

	Mappings	**Meta-Facts**	**Rules**	**Java Methods**
Running Example	4	4	10	6
Fire Detection	1	0	3	3
Temperature Monitoring	0	0	4	6
River Monitoring	0	0	5	7
Range Coverage	0	1	6	7
Average/Component[1]	0,3	0,4	2,9	3,3

Table 5.2: Example Applications: Total Component Code

As can be observed in Table 5.2, the average CrimeSPOT component has $0,3$ event-fact mappings, $0,4$ meta-facts, $2,9$ rules, and $3,3$ Java methods, which corresponds to a relatively small amount of source lines of code. While these numbers are quite application-specific, we believe that they give a fair indication of the expressiveness of our language. Implementing the same behavior in plain Java on top of event-based middleware would imply a substantial increase in component code. Such an ad-hoc implementation requires code for *event dispatching*, *event storage and management* and *event matching*. Worse, this code would be tangled with the component's application logic, rendering the latter less clear and adaptable. Furthermore, it would be duplicated across each component.

Complementary to the statistics relevant to CrimeSPOT, Table 5.1 presents general statistics relevant to Macro-CrimeSPOT. The average amount of components per application is $2,2$ and the example applications used at most 4 macro-definitions and 5 meta-variables. Obviously, using macro-definitions provides an increasing benefit as the number of developed components grows and more code can be reused. Analogously, the benefit of using application-level CrimeSPOT- and Java code (i.e., CrimeSPOT- and Java code to be added to several components in the application; specified by quantifying a code block over multiple components) increases as the number of components in an application

[1] This average was computed by first taking the average per application and afterwards averaging over the applications.

grows. On the other hand, a larger number of components per application might render Macro-CrimeSPOT's approach of centralized programming less useful as an application's source file would become larger. We plan to investigate how we can better modularize this code in future work.

5.3 Conclusion

This chapter validated the CrimeSPOT and Macro-CrimeSPOT programming languages through the implementation of illustrative example applications. As could be observed, developing WSN applications using Macro-CrimeSPOT is quite straightforward. Making a component participate in an interaction is as simple as adding a new interaction rule to its Rule Base. Furthermore, when compared to implementing an event handler to react to events, the interaction rules scale very well while keeping the application logic uncomplicated.

Next to the interaction rules, the facts also alleviate a developer's work significantly. A developer no longer has to manually store or remove events. It suffices to specify how events are reified as facts and which facts subsume which other facts, and CrimeSPOT will take care of the rest. As required, facts will be asserted in and retracted from the Fact Base, and interaction rules will be activated and deactivated.

Appendix B further evaluates our CrimeSPOT prototype implementation in terms of memory footprint, network overhead and performance. The next chapter will conclude this dissertation by discussing its contributions and future work.

6 Conclusion

Sense-and-react applications are an emerging breed of wireless sensor network applications in which certain nodes steer actuators and the processing of sensor data is typically moved inside the network. The development of these applications requires a high degree of control over the individual nodes and support for programming these nodes' interactions.

This dissertation described our work carried out in the domain of wireless sensor networks. The main objective of this work was to provide language support for programming interactions among WSN nodes to complement existing node-centric programming models. As illustrated in Chapter 2, most contemporary node-centric programming models do not help in processing data upon reception. Instead, they typically force the programmer to process this data in an event handler, even though event handlers have already been shown to violate several software engineering principles, including composability, scalability and separation of concerns [12]. One exception is FACTS, which allows the use of declarative rules for specifying interactions among WSN nodes. This approach was an inspiration for the CrimeSPOT language that was presented in the first part of this thesis (cfr. Chapter 3). Next to presenting the CrimeSPOT language, we also implemented a runtime prototype as a proof of concept (cfr. Appendix A) and presented a preliminary evaluation of its overhead (cfr. Appendix B).

The second part of this thesis focussed on a network-centric programming approach. Inspired by the state of the art, we extended CrimeSPOT with a network-centric variant, dubbed Macro-CrimeSPOT, which provides facilities for code reuse within and among WSN applications. As a proof of concept, we implemented a precompiler for compiling Macro-CrimeSPOT code to node-level CrimeSPOT code that targets our runtime prototype (cfr. Chapter 4).

In the last part of this thesis, both CrimeSPOT and Macro-CrimeSPOT were validated through illustrative examples (cfr. Chapter 5).

6.1 Contributions

The main contributions of this dissertation are:

- The introduction of the CrimeSPOT node-centric programming language that advocates the use of declarative interaction rules to specify a node's interactions.
- The introduction of the Macro-CrimeSPOT network-centric programming language and its facilities for code reuse within and among WSN applications.
- The proof of concept implementations for both the CrimeSPOT runtime and the Macro-CrimeSPOT precompiler.
- Extensions to the well-known Rete algorithm to support CrimeSPOT's meta-facts and time-related functionality (e.g., the expiration of facts and matches).

The CrimeSPOT language was based on FACTS' rule-based approach, which was described in Chapter 2 and the CrimeSPOT runtime prototype reused parts of CRIME's implementation, as described in Appendix A.

However, CrimeSPOT is significantly different from FACTS. The only aspect both approaches have in common is the use of declarative rules to specify a node's interactions. The major features present in CrimeSPOT but missing in FACTS are: the ability to use variables in interaction rules, thereby allowing a sensor node to react to several *related* facts, and the ability for a sensor node to react to the invalidation of an interaction rule's conditions.

Furthermore, CRIME's implementation did not readily translate to the setting of wireless sensor networks. As discussed in Appendix A, CRIME only provided an initial implementation for CrimeSPOT's Inference Layer. This implementation was entirely ported to the SunSPOT platform and was moreover adapted to support, amongst others: CrimeSPOT's time-related functionality (e.g., the expiration of facts and matches), the invocation of methods from within a rule's body and the association of meta-attributes with facts.

6.2 Future Work

There are several opportunities to improve on our work:

- ***Extending the CrimeSPOT language***
  - **Fact Base Sharing:** While CrimeSPOT targets event-based *component* middleware, it offers no facilities to share a Fact Base among all components on the same node. As such sharing might facilitate the interactions between node-local components, it might be interesting to further investigate this feature.
  - **Rule Priorities:** Currently, the programmer is given only limited control over the interaction rules' priorities. When triggered, interaction rules are processed in their installation order. More control over these priorities can, for instance, be given to the programmer by allowing to associate explicit priorities with rules.

  - **Control over the truth maintenance system:** Due to CrimeSPOT's truth maintenance behavior, every rule is deactivated when it loses a match. While this causes no issues for rules with an *Action* in their head, it might for rules with a fact in their head. For the latter, the programmer is given no control over the behavior that corresponds to the deactivation of the rule; the fact that was asserted earlier will simply be retracted. As this is not always the desired behavior, it might be useful to allow a programmer to disable the truth maintenance for certain rules. An example of a rule for which this would be appropriate is one that reacts to `getTemperature` requests by publishing a `temperature` fact. While the fact representing the request should be short-living, the answered `temperature` fact should not be retracted when the request expires.
  - **Fact Consumption:** While currently unsupported, it might be interesting to investigate how a rule can *consume* facts matching its conditions (i.e., hide them from other rules). One idea is to introduce a special `consume` meta-attribute for logical conditions. This meta-attribute can then specify that a fact matching the condition is consumed by the rule either as soon as it matched the condition, or as soon as the rule was triggered.
  - **Context-aware rules and meta-facts:** Certain rules or meta-facts might be context-aware, meaning that they should only be active when a certain condition holds. To support such context-aware rules or meta-facts, an additional body can be introduced for specifying the conditions for the rules or meta-facts to be active. As an example of a context-aware meta-fact, consider one specifying that certain facts should be dropped upon reception only if these facts will be processed by other nodes in the WSN.
  - **Fact publication in the n-hop neighborhood:** Currently, CrimeSPOT does not offer facilities to publish a fact in a node's physical n-hop neighborhood. The publication of facts carrying the *to(MAC=any)* meta-attribute is assumed to be network-wide. Even though the scope of fact publication highly depends on the underlying event-based middleware's capabilities, it might be interesting to investigate how CrimeSPOT can incorporate facilities to publish facts in an n-hop neighborhood. Such facilities would allow CrimeSPOT to be used to implement a wider range of applications.

- ***Default semantics for the CrimeSPOT language:*** Even though we have a rationale for the design choices in the current semantics of the CrimeSPOT language, further experiments are required to decide on a good default semantics. For instance, it might be appropriate to publish facts by default rather than requiring a `to` meta-attribute to be associated with a fact to be published. Another example of discussion is the truth maintenance behavior, which is currently enabled by default.

- ***Optimizing the CrimeSPOT prototype:*** As discussed in Appendix B, the CrimeSPOT prototype implementation is not entirely optimized for use on WSN nodes with highly-constrained resources. In particular, its memory footprint and performance can be further improved upon. For the latter, the scaffolding technique for the Rete algorithm, described by Perlin [39], can be implemented. Concretely, this implies that the causal links between facts and reactions are kept. When a fact is subsequently retracted, it becomes straightforward to identify exactly those reactions that need to be undone. As a consequence, facts can be retracted in constant time because it's no longer required to propagate a negated token through the Rete network.

  Additionally, a more efficient implementation for the Middleware Bridge can be built. Concretely, one that implements unicast fact publication by efficiently registering specific LooCI bindings at runtime.

A

CrimeSPOT Prototype Implementation

This appendix will briefly explain the highlights of the CrimeSPOT prototype implementation. The prototype is an extension to the CRIME coordination language [40] and was instantiated on top of the LooCI event-based component middleware [25] for the SunSPOT platform. As the prototype's source code is too voluminous to add to this appendix, the reader can consult it online[1]. In the following sections, we will visit all layers of the CrimeSPOT runtime (cfr. Figure 3.3) and conclude with more details on how to implement a CrimeSPOT component that uses the prototype implementation. Note that such a component is the compilation target for the Macro-CrimeSPOT language.

A.1 Inference Layer

As mentioned before, the Inference Layer consists of the Fact Base, the Rule Base and the Inference Engine. For implementing this layer, we were able to reuse a large part of the CRIME implementation.

A.1.1 The CRIME Inference Engine

CRIME [40] is a logic-based coordination language that allows applications to use logic rules to specify how they should respond to changes in the environment. These changes are modeled by the addition or removal of facts that contain context information. Because CRIME allows applications to respond to the removal of facts, it is also a truth maintenance system.

CRIME conceptually stores its facts in a Fact Base and its rules in a Rule Base. CRIME's Inference Engine (i.e., a Rete [37] forward chaining inference engine) evaluates the rules from the Rule Base against the Fact Base whenever the latter changes. Because this behavior is exactly what's required for the

[1] `http://soft.vub.ac.be/amop/crime/sunspot`

CrimeSPOT language (cfr. Section 3.4), CRIME provided a solid basis for the prototype implementation. Next to the Rete implementation, it also provided some useful constructs that can be used within the rules (i.e., the `not`, `findall`, and `length` operators, cfr. Section 3.7.1.4).

A.1.1.1 The RETE Algorithm

The most relevant part of the CRIME language is its Inference Engine, which implements the Rete algorithm [37]. Rete is an efficient forward chaining inference algorithm that optimizes its inference process by using a combination of smart caching techniques. To illustrate how the algorithm works, we can consider the CRIME rule from Listing A.1. This rule asserts `temperatureInTent` facts as soon as the necessary `temperature`, `online` and `tent` facts have been asserted in the Fact Base. Conversely, when one of these facts is retracted from the Fact Base, the rule retracts the corresponding `temperatureInTent` facts.

```
1 temperatureInTent(Celsius=?temp, Tent=?tent)
2    <- temperature(Celsius=?temp, FromMAC=?tMac),
3       online(TID=?tentId, MAC=?tMac),
4       tent(ID=?tentId, Name=?tent).
```

Listing A.1: RETE Algorithm: a rule for asserting temperatureInTent facts

The Rete algorithm transforms rules to a tree-like structure, called a Rete network. This network represents the Rule Base and has a single *root node*, in which all new facts are inserted. Because facts are stored in the network, the network also represents the Fact Base. A Rete network consists of two parts. The first part is the *alpha network*, which contains *filter nodes*, and the second part is the *beta network*, which contains *join-* and *production nodes*. Whenever a fact is asserted, it is encapsulated in a *token* that is inserted in the root node and propagated through the network. Figure A.1 depicts the Rete network after installing the example rule from Listing A.1 and asserting some facts. Note that tokens are denoted between < and > brackets.

For every condition in a rule's body, the Rete algorithm generates two filter nodes: one for checking the type of the fact in the condition and one for checking the inner constraints of the condition. These filter nodes will only allow tokens to pass if these tokens satisfy their constraints. For instance, for the `temperature(Celsius=?temp, FromMAC=?tMac)` condition, the first filter node verifies that the inserted token has a fact with the temperature type, while the second filter node verifies that this fact also has Celsius and FromMAC attributes. Note that every filter node also has a memory to cache the inserted tokens that satisfied their constraints. For instance, it can be observed in the Figure that the asserted `temperature(Fahrenheit=50)` fact passed the constraint of the temperature filter node (i.e., it has the temperature type), but it didn't pass the second filter node for the temperature condition (i.e., it doesn't have the Celsius and FromMAC attributes).

Every token that passes the second filter node enters the beta network. The top nodes in the beta network are called join nodes and will try to join an inserted token with all tokens from the memory of their other parent node (i.e., the parent that didn't insert the token). For instance, when the `<temperature(Celsius=30, FromMAC="m1")>` token is propagated to the first join node in the example, this join node will attempt to join this token with

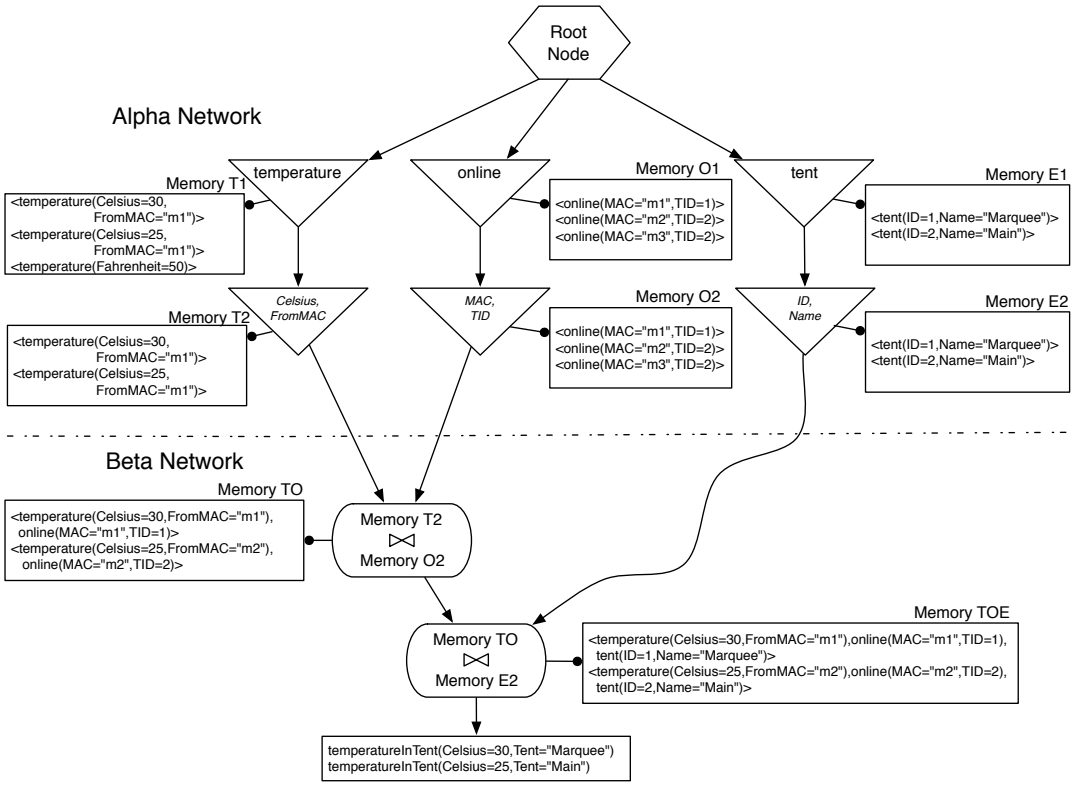


Figure A.1: A RETE Network with a single rule for asserting temperatureInTent facts

all tokens in the memory of the second online filter node. For a pair of tokens to be joinable, the variable bindings that they provide have to be consistent. In the example, this means that the token containing a `temperature` fact will only be joined with a token containing an `online` fact if these tokens provide the same binding for `?tMac`. Consequentially, the `<temperature(Celsius=30, FromMAC="m1")>` token can only be joined with the `<online(MAC="m1",TID=1)>` token. Note that every pair of joined tokens is merged into a single token, cached in the memory of the join node and further propagated through the network.

After the required propagating and joining, a token will eventually reach the bottom node in the beta network (i.e., the production node), which means that a match for all of the rule's conditions has been found and that the rule's reaction can be invoked. For instance, whenever the production node from the example rule receives a token, it will assert the corresponding `temperatureInTent` fact (i.e., the one instantiated with the token's variable bindings). Note that in this RETE network, no rule has a condition for `temperatureInTent` facts and that these facts are therefore not immediately useful.

Whenever a fact is retracted, it is also encapsulated in a token that is inserted in the root node and propagated through the RETE network. To

distinguish such a token from a regular token, it carries a negation sign while a regular token carries an addition sign. Unlike positive tokens, negated tokens are *not* stored in the RETE network. Whenever a negated token passes a node in the network, the previously stored corresponding token (i.e., the positive one containing the same facts) will be removed from that node's memory before the negated token is further propagated. When a negated token eventually reaches a production node, the rule corresponding to this production node has lost a match and will invoke its compensating reaction. For the example rule, this means that the `temperatureInTent` fact corresponding to the negated token will be retracted.

Note that the RETE algorithm gains its efficiency through state-saving. By using memories to store intermediate results in both the alpha and beta part of the network, no previously calculated results have to be recomputed when a new fact is asserted or retracted. In addition, the algorithm shares the type filter nodes among various conditions of various rules and is therefore as memory-efficient as possible.

A.1.1.2 Sequentializing Access to the RETE Network

CRIME employs a thread-safe *agenda* to which *activations* can be added. All of the agenda's activations are activated one after another in a first-in first-out order. As an example of an activation, one can consider an activation for asserting or retracting a fact. When this activation is activated, it will encapsulate the fact in a token and insert this token in the root node of the RETE network. This activation is, for instance, added to the agenda whenever a token is inserted in a production node (e.g., to assert or retract the corresponding fact) or when a new fact is received from the network.

A.1.2 Introducing Time-related Functionality

While CRIME's RETE implementation provides a good basis for CRIMESPOT's Inference Layer, it doesn't provide support for CRIMESPOT's time-related features such as *fact expiration*, *match expiration* and *match verification*. To support such functionality, we introduced a *timer* that can be used to add activations to the aforementioned agenda. This timer runs in a separate thread and allows activations to be scheduled for addition in the agenda after a particular amount of time has passed, or at every interval of a particular amount of time. As we will discuss in the following sections, the activations scheduled in the timer vary for each of the time-related features.

A.1.2.1 Dealing with Timeouts

Fact Expiration CRIMESPOT's most notable time-related feature is the expiration of facts. As mentioned before, a fact can have an associated *factExpires* meta-attribute that specifies how long this fact remains valid. Whenever a fact carrying this meta-attribute is asserted, an activation for retracting the fact will be scheduled in the timer. This activation will be added to the agenda after the amount of time that was specified in the *factExpires* meta-attribute has passed, and the fact will consequentially be retracted.

In addition, *fact expiration* also required the introduction of *fact IDs* to uniquely identify facts. There were two reasons for this requirement. First of all, it's perfectly possible to assert the same fact multiple times, in which case the conceptual Fact Base contains two or more identical facts. Secondly, a fact carrying a *factExpires* meta-attribute can be retracted *before* it has expired (e.g., when the rule that asserted it has lost a match). When this happens, the scheduled retraction activation for this fact should be unscheduled. However, when there can be multiple retraction activations for several identical facts, it becomes impossible to decide which activation to unschedule, even though all these activations are fundamentally different because they will be added to the agenda at different times. The rule in Listing A.2 further illustrates this problem. This rule asserts a `gotReading()` fact whenever a `reading` fact is asserted.

```
1 gotReading()@[factExpires(Seconds=999)]
2    <- reading(T=?v).
```

Listing A.2: A rule for motivating fact IDs

When `reading(T="x")` is asserted at time $t1$, and `reading(T="x")` is asserted at time $t2$ (with $t2 > t1$), `gotReading()` will be asserted twice and scheduled for retraction at time $t1 + 999$ and at time $t2 + 999$. Now, when at a later time $t3$, the last `reading(T="x")` gets retracted, the last `gotReading()` should also be retracted and its retraction activation should be unscheduled. However, it's impossible to decide which retraction activation for `gotReading()` to unschedule: the one that will be added to the agenda at $t1 + 999$ or the one that will be added at $t2 + 999$. Associating a unique *fact ID* with every fact solves this issue because this ID also uniquely identifies a fact's retraction activation (if any), which can therefore easily be unscheduled when a fact is retracted.

Match Expiration Another time-related feature is *match expiration*. This feature is employed whenever a logical condition in a rule's body carries a *matchExpires* meta-attribute. This meta-attribute specifies how long a match for the associated condition remains valid (cfr. Section 3.7.1.3). Note that, in RETE terms, a match for a condition corresponds to the insertion of a positive token in the condition's second filter node (i.e., the one that checks the condition's inner constraints), while the invalidation of a match for a condition corresponds to the insertion of a negated token in this node.

To introduce match expiration, we extended the aforementioned *filter nodes* from the RETE network with so-called *filter actions*. A filter action represents additional logic to be invoked whenever a valid token is inserted in its filter node. Hence, whenever a filter node receives a token that satisfies its constraints, it will not only store and forward this token, but it will also invoke all of its associated filter actions with this token. For every logical condition carrying a *matchExpires* meta-attribute, a special filter action will be added to the filter node that checks the condition's inner constraints (i.e., the condition's second filter node). Whenever this filter action is invoked with a positive token, it will schedule a *token insertion activation* in the timer, which will be added to the agenda after the amount of time that was specified in the *matchExpires* meta-attribute has passed. This activation will insert a negated copy of the previously inserted token in the filter node, which will consequentially be propagated through the RETE network and make the match (that was represented by the positive token)

expire. Because a condition can also lose a match due to the retraction of a fact (i.e., before the match *expired*), the aforementioned filter action will, whenever it is invoked with a negated token, unschedule the previously scheduled *token insertion activation* because the match no longer has to expire.

Match Verification Finally, the *match verification* feature is also time-related. This feature is employed whenever a logical condition in a rule's body carries a *matchEvery* meta-attribute. A *matchEvery* meta-attribute specifies the expected matching for its associated condition and will cause a `timedOut` fact to be asserted if the condition isn't matched in the specified amount of time (cfr. Section 3.7.1.3).

This feature is also implemented using the aforementioned *filter actions*. For every logical condition carrying a *matchEvery* meta-attribute, a special filter action will be added to the filter node that checks the condition's inner constraints (i.e., the condition's second filter node). Upon initialization, this filter action schedules a *timeout activation* in the timer that will be added to the agenda after the amount of time that was specified in the *matchEvery* meta-attribute has passed. When this activation is activated by the agenda, it will assert the `timedOut` fact for the condition. Whenever the filter action is invoked with a positive token (i.e., whenever the condition obtained a new match), it will verify whether the condition already timed out. If it did, the filter action will retract the `timedOut` fact and reschedule the *timeout activation* in the timer such that it is added to the agenda after the amount of time that was specified in the *matchEvery* meta-attribute has passed starting from this point in time. If it didn't, the filter action will only reschedule the *timeout activation*.

A.1.3 Introducing a Compute Node for the RETE Network

Next to the *logical conditions* for matching facts in the body of a rule, CRIMESPOT also supports *extra-logical conditions*. Extra-logical conditions employ operators and obtain matches by invoking predefined functionality (cfr. Section 3.7.1.4). While some of CRIMESPOT's extra-logical conditions are a part of CRIME (i.e., the `not`, `findall`, and `length` operators), the `is` operator is not. As discussed earlier, this operator can be used to invoke one of the component's methods from within the body of a rule. The syntax for an is-expression (i.e., a condition that employs the `is`-operator) is shown below. Note that the `is`-operator will only provide a match if the expression on its

⟨*is_expression*⟩ ::= (⟨*value*⟩ | ⟨*variable*⟩) `is` ⟨*methodinvocation*⟩

⟨*methodinvocation*⟩ ::= ⟨*name*⟩`(` [⟨*argument*⟩ (`,` ⟨*argument*⟩)*] `)` [`@[`⟨*option*⟩`]`]

⟨*argument*⟩ ::= ⟨*value*⟩ | `(`(⟨*typecast*⟩)⟨*variable*⟩

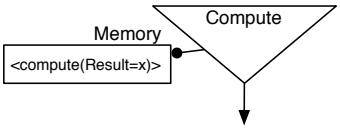
⟨*typecast*⟩ ::= `Number` | `Fraction` | `Text` | `Symbol` | `List`

⟨*option*⟩ = `evalEvery(Seconds=`⟨*number*⟩`)`
| `renewEvery(Seconds=`⟨*number*⟩`)`

left-hand side can be unified with the result of the method invocation on its right-hand side.

For every is-expression in a rule's body, a *compute node* is created (i.e., to invoke the method) and fitted in the RETE network. When the is-expression is the rule's first condition, the *compute node* will not have a parent, while otherwise, it will have a parent. To simplify the explanation of this node, we will distinguish between these two cases.

A Compute Node without a parent

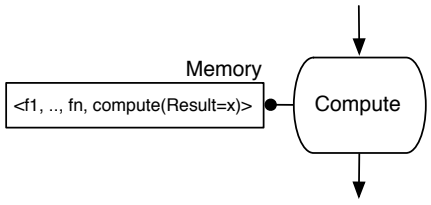


A compute node without a parent can be compared to a filter node. It will filter the results of method invocations and will only allow the results that unify with the expression on the left-hand side of the is-operator to pass.

Unlike a filter node, a compute node is not connected to the RETE network's root node. This is because it doesn't rely on the assertion or retraction of facts to do its work. As soon as the compute node is initialized (i.e., fitted in the RETE network), it will invoke its method and verify whether the result can be unified with the expression on the left-hand side of the is-operator. If this unification succeeds, the result is placed in a special fact that is encapsulated within a token. This token is stored in the compute node's memory and propagated to its child node. Note that this special fact will never be asserted in the Fact Base and can therefore not be matched in other conditions. Its sole purpose is to represent the result of the compute node (e.g., to bind a variable, when the expression on the left-hand side of the is-operator was an unbound variable).

When the is-expression specified a scheduling option for the method invocation, the compute node will invoke its method multiple times. In this case, a *compute activation* will be scheduled in the timer that will be added to the agenda at every interval of the time specified by the option (e.g., every 10 seconds). This activation will trigger the compute node. When the *evalEvery* option was employed, the compute node will just repeat the aforementioned behavior when it is triggered. On the other hand, when the *renewEvery* option was employed, the compute node will first remove the stored token from its memory, negate it, and forward it to its child node (i.e., to invalidate the previous match) before repeating the aforementioned behavior.

A Compute Node with a parent.



A compute node with a parent can be compared to a join node. It will join every inserted token with the result of the corresponding method invocation.

Due to the presence of a parent, the compute node will do its work as soon as a token has been inserted. When a *positive token* is inserted, the node

will use this token to invoke its method (e.g., to extract values for variable method arguments) and to verify the unifiability of the invocation's result and the (possibly variable) expression on the left-hand side of the is-operator. If the unification succeeds, the node will merge the inserted token with the token containing the invocation result. The resulting token is stored in the compute node's memory and propagated to its child node.

On the other hand, when a *negated* token is inserted, the method should not be re-invoked but the previously propagated positive token should be invalidated. To this end, the node will search its memory for the token with the result of the method invocation that employed the positive token corresponding to the inserted negated token (i.e., the positive token with the same facts). This token will be removed from its memory, negated, and propagated to its child node.

When the is-expression specified a scheduling option for the method invocation, the compute node will invoke its method multiple times for every inserted *positive token*. Therefore, for every such token, a *compute activation* is scheduled in the timer that will be added to the agenda at every interval of the time specified by the scheduling option. Unlike the aforementioned *computation activation*, this activation will trigger the compute node *using the token*. When the *evalEvery* option was employed, the compute node will just behave as if a new positive token is inserted when it is triggered. On the other hand, when the *renewEvery* option was employed, the compute node will first remove the stored token with the result of the previous method invocation for the token from its memory, negate it, and forward it to its child node (i.e., to invalidate the previous match) before behaving as if a new positive token is inserted. When a *negated token* is inserted in the compute node, the method will no longer have to be invoked for the corresponding positive token and therefore, this positive token's *compute activation* is unscheduled.

A.2 Reification Layer

As explained earlier, the reification layer consists of the Reification Engine and the Configuration Base. While its main purpose is to reify events as facts upon reception (i.e., to be asserted in or retracted from the Fact Base), and to reify facts as events (i.e., to be published through the underlying middleware), it also takes care of processing the subsumption- and drop meta-facts that are conceptually present in the Configuration Base (cfr. Section 3.8). In this section, we will explain how these meta-facts are implemented.

A.2.1 Processing Incoming Events

The conceptual processing of incoming events has already been illustrated in the Event reification pipeline (cfr. Figure 3.15). When the Reification Engine receives an event, it will first reify it as a fact and verify whether it should be dropped. If the fact shouldn't be dropped, the Reification Engine will retract all facts that were subsumed by the newly received fact, after which it will assert the new fact in the Fact Base. To implement this functionality, three activations are added to the aforementioned agenda after an event was reified as a fact: one for verifying whether the fact should be dropped, one for retracting all

facts that were subsumed by the newly received fact, and one for asserting the new fact in the Fact Base. While the activation for asserting a fact is by now straightforward, the following sections will clarify what the other activations actually do.

A.2.2 Implementing the Meta-Facts

While meta-facts are conceptually represented as entities stored in the Configuration Base, their functionality is implemented as special-purpose RETE networks. These networks are rooted in the Reification Layer and employ special-purpose filter-, join- and production nodes. In addition, they are joined with the previously explained RETE network from the Inference Layer.

A.2.3 Building a RETE Network for Drop Meta-Facts

Whenever a drop meta-fact is added to the CRIMESPOT runtime, the Reification Layer's RETE network for drop meta-facts is extended. To this end, the drop meta-fact is transformed as if it was a rule. To illustrate this transformation, we can consider the following drop meta-fact:

```
1 drop!(Incoming=temperature(Celsius=?n)@[from(MAC=?m)],
2       When=[refuseTemp(From=?m)].
```

This meta-fact specifies that a newly received `temperature` fact should be dropped when there's a `refuseTemp` fact in the Fact Base for the sensor node that published the `temperature`. Figure A.2 depicts the RETE network after installing the above drop meta-fact. While the drop meta-fact's body (i.e., the value of the *When* attribute) is transformed like a regular rule's body and added to the Inference Layer's RETE network, the incoming fact's specification (i.e., the value of the *Incoming* attribute) is transformed like a regular condition and added to the Reification Layer's RETE network for drop meta-facts. Note that the filter-, join- and production nodes in the Reification Layer's *drop* RETE network are different than the previously discussed ones. First of all, the filter- and join nodes do not cache any tokens. Furthermore, while the filter node behaves the same, the join node will only try to join an inserted token from the second temperature filter node with the tokens in memory `R2` (and not the other way around). As soon as one pair of tokens was joined, the join node will not attempt to join the inserted token with any other token from memory `R2`. It will immediately propagate the merged token to the production node. Finally, when a token reaches the production node, this node will immediately invoke its functionality (i.e., rather than scheduling an activation in the agenda).

When an activation for verifying whether a received fact should be dropped is activated[2], the received fact is encapsulated in a token that is inserted in the root node of the Reification Layer's network for drop meta-facts. As an example, consider the reception of a `temperature(Celsius=-1)@[from(MAC="m2")]` fact. The token encapsulating this fact will be passed by both the first and the second filter node in the *drop* network because it has the temperature type and a Celsius attribute. When it reaches the join node, the join node will be able to join it with `<refuseTemp(From="m2")>` and will propagate

[2]Remember, this activation was added to the agenda by the Reification Engine after it reified a received event as a fact.

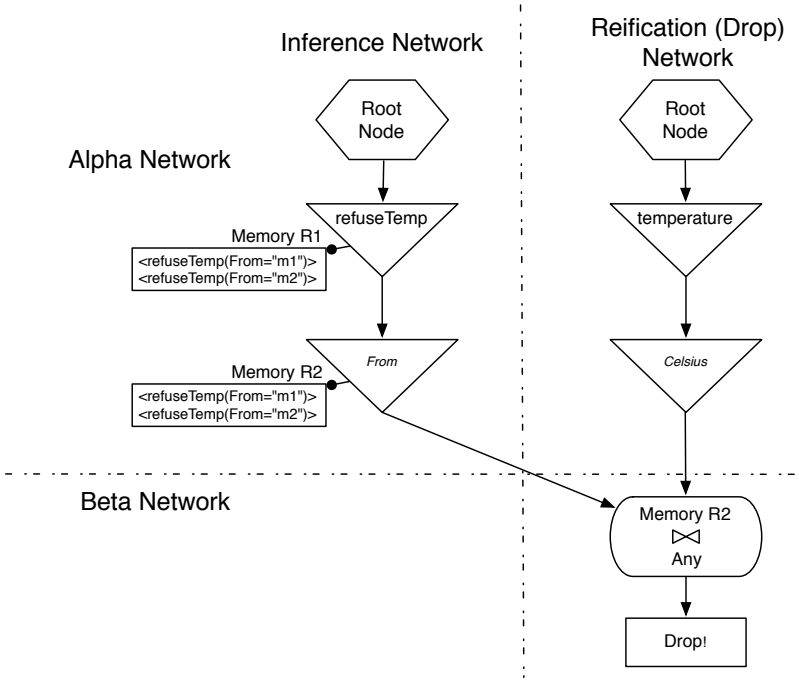


Figure A.2: A RETE Network with a single meta-fact for dropping temperature facts

`<refuseTemp(From="m2"),temperature(Celsius=-1)@[from(MAC="m2")]>` to the production node. The latter will immediately invoke its functionality: as the received fact should be dropped, it will remove both the activation for retracting all facts that were subsumed by the received fact, and the activation for asserting the received fact from the agenda.

If we consider the reception of a `humidity(Percent=50)@[from(MAC="m2")]` fact, this fact's token will never pass the first filter node and therefore, the remaining activations for dealing with subsumptions and for asserting the fact in the Fact Base will not be removed from the agenda but will just be activated one after another as soon as the token propagation stopped.

Note that this *drop* RETE network introduces a minimal overhead when a token is inserted in its root node. All required results in the Inference Layer's network are cached and don't have to be recomputed.

A.2.4 Building a RETE Network for Subsumption Meta-Facts

Whenever a subsumption meta-fact is added to the CRIMESPOT runtime, the Reification Layer's RETE network for subsumption meta-facts is extended. To this end, the subsumption meta-fact is transformed as if it was a rule. To illustrate this transformation, we can consider the following subsumption meta-fact:

```
1 subsumes!(Incoming=temperature(Celsius=?new)@[from(MAC=?m)],
```

```
2           Fact=temperature(Celsius=?old)@[from(MAC=?otherm)],
3           When=[online(Tent=?tnt)@[from(MAC=?otherm)],
4                 online(Tent=?tnt)@[from(MAC=?m)]]).
```

This meta-fact specifies that a received `temperature` fact subsumes all existing `temperature` facts that originate from the same tent. Figure A.3 depicts the RETE network after installing the above subsumption meta-fact. Before

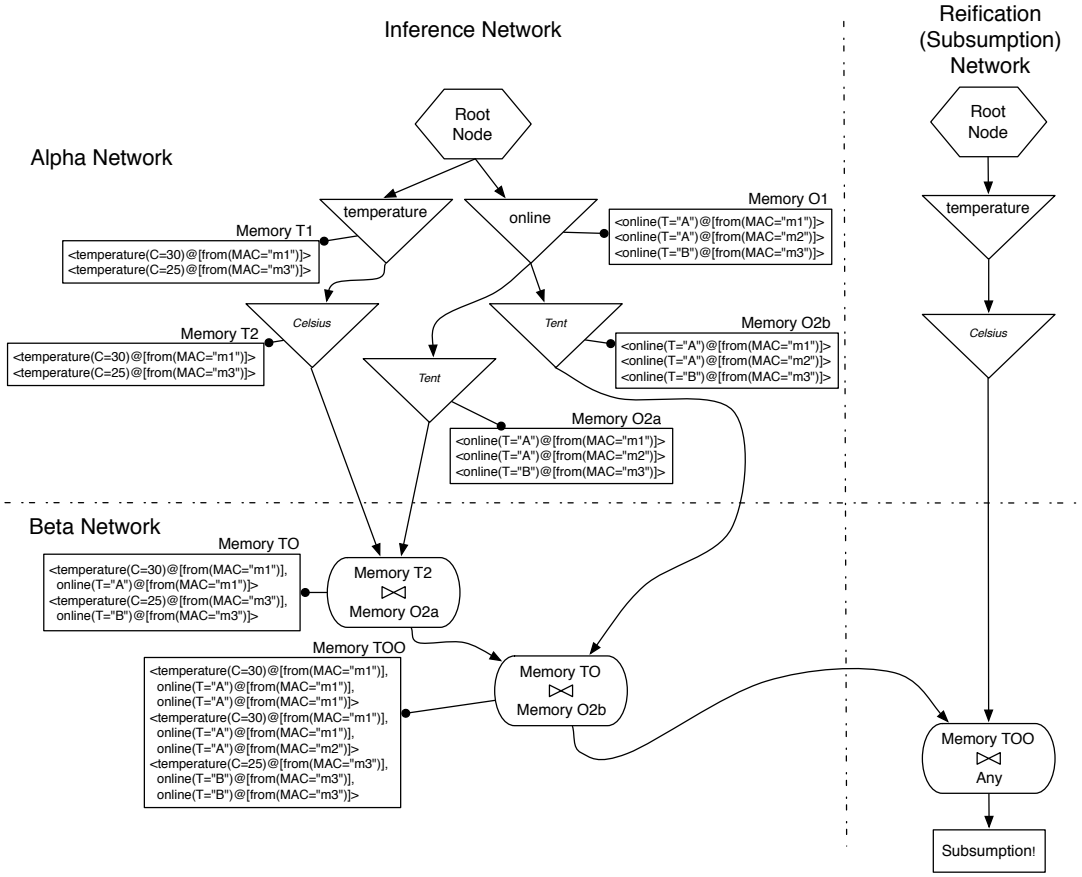


Figure A.3: A RETE Network with a single meta-fact that specifies the subsumption strategy for temperature facts

transforming the subsumption meta-fact's body (i.e., the value of the *When* attribute), it is merged with the fact-to-subsume's specification (i.e., the value of the *Fact* attribute). For instance, the above meta-fact's body is actually: `temperature(Celsius=?old)@[from(MAC=?otherm)], online(Tent=?tnt) @[from(MAC=?otherm)], online(Tent=?tnt)@[from(MAC=?m)]`. This body will be transformed like a regular rule's body and added to the Inference Layer's RETE network[3]. The incoming fact's specification (i.e., the value of the *Incoming* attribute), on the other hand, is transformed like a regular condition and added

[3]Note that the above network also illustrates the sharing of type filter nodes: the type filter node for `online` facts is shared among both `online` conditions in the subsumption meta-fact's body.

to the Reification Layer's RETE network for subsumption meta-facts. Most nodes in this *subsumption* network are identical to the ones of the aforementioned *drop* network. However, the join node will attempt to join any inserted token from the second temperature filter node with *all* tokens from memory `TOO` and will consequentially propagate *all* successfully joined tokens to its child (i.e., the production node). Note that this network construction allows the production node to easily identify the subsumed facts: a subsumed fact will be in the first position of an inserted token.

To illustrate how the above network functions, we can now consider the activation for retracting all facts that were subsumed by a newly received fact[4]. When this activation is activated, the newly received fact is encapsulated in a token that is inserted in the root node of the Reification Layer's network for subsumption meta-facts. As an example, consider the reception of a `temperature(Celsius=28)@[from(MAC="m2")]` fact that wasn't dropped. The token encapsulating this fact will be passed by both the first and the second filter node in the *subsumption* network because it has the temperature type and a Celsius attribute. When it reaches the join node, the join node will be able to join it with `<temperature(Celsius=30)@[from(MAC="m1")], online(Tent="A")@[from(MAC="m1")], online(Tent="A")@[from(MAC="m2")]>` and will propagate `<temperature(Celsius=30)@[from(MAC="m1")], online(Tent="A")@[from(MAC="m1")], online(Tent="A")@[from(MAC="m2")], temperature(Celsius=28)@[from(MAC="m2")]>` to the production node. The latter will immediately invoke its functionality: because a token was inserted, an incoming fact subsumed another fact (i.e., the one in the first position of the inserted token: `temperature(Celsius=30)@[from(MAC="m1")]`), and this subsumed fact is consequentially retracted from the Fact Base.

Note that, just like the *drop* RETE network, the *subsumption* network introduces a minimal overhead when a token is inserted in its root node. All required results in the Inference Layer's network are cached and don't have to be recomputed.

A.3 Infrastructure Layer

The Infrastructure Layer binds the CRIMESPOT runtime to the underlying middleware. To this end, a middleware-specific implementation for the Middleware Bridge should be provided which transfers the events received from the event-based middleware to the Reification Engine. In addition, the Middleware Bridge should also implement all required middleware-specific functionality, which will be invoked by the Reification Engine.

For CRIMESPOT's prototype implementation, we used the LOOCI event-based component middleware for the SunSPOT platform and instantiated the middleware bridge accordingly. This section will introduce the LOOCI middleware and give more details regarding the implementation of its middleware bridge.

[4]Remember, this activation was added to the agenda by the Reification Engine after it reified a received event as a fact.

A.3.1 LooCI

The LooCI [25] middleware introduces both a *component infrastructure* and a *decentralized event-bus* to wireless sensor networks. It allows multiple components to be deployed on the same WSN node and it allows these components to communicate by exchanging events through the decentralized event-bus. Components can both publish events and subscribe to certain event types. Based on the event-subscriptions (i.e., *wirings* in the LooCI jargon), the events published in the WSN application will be routed from the publishers to the subscribers through the decentralized event-bus. This is illustrated in Figure A.4 (taken from the LooCI website[5]), which depicts a runtime view on a WSN node running LooCI. Next to the components and the event-bus, it also depicts the Re-

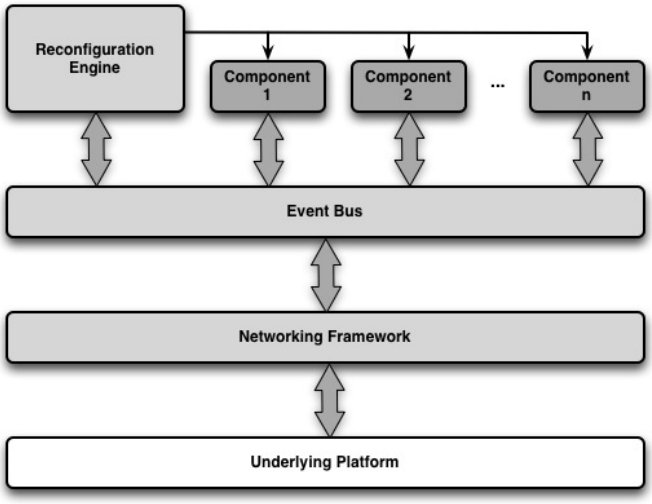


Figure A.4: Runtime view on a WSN node running LooCI

configuration Engine, which is the heart of the LooCI middleware and controls the components running on the WSN node. To allow a developer to control the Reconfiguration Engine, LooCI comes with a *gateway* application. This gateway should run on a back-end entity and can, for instance, be used to remotely deploy components on the nodes in the wireless sensor network.

A.3.2 Instantiating the Middleware Bridge for the LooCI Middleware

To instantiate the Middleware Bridge, an implementation for the following methods has to be provided:

- `void subscribeToEvent(EventDescription desc)` subscribes to the events of the given `Event Description`. This method is called whenever a new event-fact mapping has been added to the CrimeSPOT runtime.

[5] `http://code.google.com/p/looci/`

- `Object toEvent(GenericFact fact, Mappings m, Boolean isAssertion)` converts the given `fact` to an event and returns it.
- `void publish(Object event)` publishes the given `event`.
- `GenericFact toFact(Object event, Mappings m)` converts a received `event` to a `GenericFact` and returns it. This method can return `null` if the `event` is misaddressed, which is useful for middleware that doesn't allow directed event publications.
- `Boolean isAssertion(Object event, Mappings m)` decides whether the `event` represents a fact to be asserted or retracted.

Given the LooCI API, the implementation for these methods is quite straightforward. We assigned a specific event type for publishing non-mapped facts and for publishing fact retractions and subscribed each CrimeSPOT component to these event types (i.e., by registering so-called *toAll* and *fromAll* wirings with LooCI's Reconfiguration Engine). All of the mapped facts or events are converted to events or facts using the mappings. Note that LooCI does *not* support directed event publications and that this functionality is therefore simulated in our middleware bridge implementation by verifying an event's destination at the receiver.

Subscribing to events Using the LooCI API, it is possible to subscribe a component to a specific event type at runtime by registering so-called *toAll*- and *fromAll* wirings with the Reconfiguration Engine. The implementation of the `subscribeToEvent(EventDescription)` method registers these wirings whenever it is called (i.e., after an event-fact mapping was added to the CrimeSPOT runtime).

Converting Facts to Events Depending on whether the `toEvent` method is called to create an event for a fact assertion or a fact retraction, the `fact` is converted to a potentially mapped event or a retraction event (i.e., an event of the aforementioned fact retraction type).

In the case of a fact assertion, the `Mappings` dictionary is first searched for a mapping for this fact. If such a mapping is found, the `fact` is converted to the corresponding event. In addition, extra fields are added to this event's payload to store the fact's ID and destination (i.e., the destination MAC address and component ID). As LooCI does not allow directed publications, such publications are simulated: the extra destination fields will be read in the `toFact` method to decide whether or not to accept a received event. Regular LooCI components will typically not read these extra fields. On the other hand, if no mapping is found, an event of the aforementioned event-type for non-mapped facts is created. This event has, as its payload, a single String containing the fact's textual representation.

In the case of a fact retraction, an event of the event-type for fact retractions is created. This event has, as its payload, a String containing the network-unique identifier for the fact (i.e., a combination of the originating component's MAC address, its component ID, and the fact ID). In addition, it also has extra fields

containing the destination component's MAC address and component ID (i.e., to simulate a directed publication).

Publishing Events The `publish(Object)` method maps exactly on the `publish(Event)` method that's provided by the LooCI API.

Converting Events to Facts Converting an event to a fact is analog to the inverse process.

In the case the event type is present in the `Mappings` dictionary or is equal to the non-mapped fact event type, the event is converted to a fact accordingly. To convert a fact's textual representation to a fact (i.e., in the case of a non-mapped fact event type), the CrimeSPOT runtime provides a conversion method to parse the textual representation. Misaddressed events are detected by comparing the fact's destination address (if any), with the current component's address.

Else, if the event type is of the fact retraction type, the corresponding fact is obtained from the CrimeSPOT runtime by querying the Fact Base using the fact's unique identifier.

Deciding on fact assertion or retraction Due to the above implementation, the implementation of `isAssertion` is simple. It suffices to verify whether the event carries the fact retraction event type to decide whether a given event represents a fact assertion or -retraction.

A.4 Macro-CrimeSPOT's Compilation Target

As we've explained earlier, Macro-CrimeSPOT code compiles to node-level component code (cfr. Section 4.3). This node-level code employs the CrimeSPOT prototype implementation that was discussed in this appendix. In this section, we will show what this node-level code looks like, or, how a component can be implemented by using only the CrimeSPOT language.

When using the CrimeSPOT prototype implementation, a component can be implemented by extending the `CrimeSPOTComponent` class. A component has to provide an implementation for two methods: the `execute()` method, which will be invoked when the component is started, and the `invoke(String,Object[])` method, which is a *dispatcher method* for dispatching method names to method invocations. An implementation for the dispatcher method is required to allow the CrimeSPOT runtime to invoke methods on the component in the absence of reflection on the SunSPOT platform.[6] For every method that's invoked from within an interaction rule's body, there should be an entry in this dispatcher method.

Listing A.3 depicts an example component's implementation in Java. This component publishes `temperatureReading` facts and reacts to `adjustHeating` facts by steering its associated heater accordingly.

[6]Because interaction rules are added to the CrimeSPOT runtime at runtime, the methods to invoke from within these rules' bodies are also only known at runtime. Dynamically invoking methods at runtime requires some form of reflection, which is not available on the SunSPOT platform.

```
1 public class ExampleComp extends CrimeSPOTComponent {
2   public ExampleComp() { super("Example Component"); }
3
4   public void execute() {
5     processExpression("temperatureReading(Celsius=?temp)@[factExpires(Seconds=600)," +
6                                                          "to(MAC=\"any\")]" +
7                       "  <- ?temp is getTemperature()@[renewEvery(Seconds=600)].");
8
9     registerRule("adjustHeating(Level=?h)",this.adjustHeater);
10  }
11
12  private Action adjustHeater = new Action() {
13    public void activated(TypedObject arguments) { /* adjust heater */ }
14    public void deactivate(TypedObject arguments) { /* reset heater */ }
15  };
16
17  public Attribute invoke(String method, Object[] args){
18    if(method.equals("getTemperature__"))
19      return getTemperature();
20    else
21      return ExpressionFactory.attribute("ERROR: Method not in the dispatcher!");
22  }
23 }
```

Listing A.3: Macro-CrimeSPOT's compilation target: a CrimeSPOT Component in Java

Note that, because the `getTemperature()` method is invoked from within an interaction rule's body, there's an entry for this method in the dispatcher method. Its implementation, however, is not required in this class. The `getTemperature()` method is part of the predefined I/O methods provided by the `CrimeSPOTComponent` class. Other predefined I/O methods are: `getMac()`, `getLight()`, `getAcceleration()` and `turnOnLeds(amount,color)`.

For a component of a Macro-CrimeSPOT application, both the `execute` and `invoke` methods are implemented by Macro-CrimeSPOT's precompiler. However, to allow a Macro-CrimeSPOT programmer to initialize her component using Java logic, the `initialize()` method can be overridden in a Java code block. This method is invoked whenever a component is started.

A.5 Conclusion

In this appendix, we presented the most important aspects of the CrimeSPOT prototype implementation[7] by discussing the highlights of its three layers. For the Inference Layer, CRIME's Inference Engine (i.e., a Rete engine) was partly reused, yet also extended. We covered the most important extensions, which were required for CrimeSPOT's time-related functionality (i.e., *fact expiration*, *match expiration* and *match verification*) and for propagating the results of method invocations through the Rete network. The Reification Layer was entirely built from scratch. The most notable aspects of this layer were the special purpose Rete networks for implementing CrimeSPOT's meta-facts. Finally, we also discussed the interface for the Infrastructure Layer's Middleware Bridge. This interface should be implemented for specific event-based WSN middleware in order to use the CrimeSPOT prototype on top of it. As an

[7]The source code can be consulted at `http://soft.vub.ac.be/amop/crime/sunspot`.

example, we showed how the Middleware Bridge can be instantiated for the LooCI event-based component middleware.

Next to discussing the CrimeSPOT prototype implementation, we also showed how this prototype can be used. While implementing a CrimeSPOT component is simple, the implementation of the *dispatcher* method can be considered cumbersome. However, using Macro-CrimeSPOT and its accompanying precompiler relieves a programmer from this burden.

In the next appendix, we will evaluate our CrimeSPOT prototype implementation in terms of memory footprint, network overhead and performance.

B

Evaluating the CrimeSPOT Prototype

Having explained the CrimeSPOT prototype implementation, we will now give a preliminary evaluation of its overhead. All required experiments were conducted on standard SunSPOT motes (180MHz ARM9 CPU, 512KB RAM, 4MB Flash, SQUAWK VM version 'RED-100104') running the LooCI middleware version 1.0 beta (20101011).

B.1 Memory Footprint

We measured CrimeSPOT's memory footprint both in terms of ROM and RAM.

Table B.1 summarizes the results for the ROM usage. Clearly, the

	LooCI	**CrimeSPOT**	**Average Component**
ROM	52,14kB	406,66kB	1,99kB

Table B.1: Memory Footprint (ROM)

CrimeSPOT implementation consumes a considerable amount of Flash memory (i.e., 9,7% of the SunSPOT's Flash). We expected this result as our prototype implementation includes the CRIME implementation which is not yet entirely cleaned up and optimized for use on sensor motes with small Flash memory. Further cleaning up the implementation is left for future work. However, like the LooCI middleware, the CrimeSPOT prototype has been included in the SunSPOT library and hence should never be remotely deployed using the LooCI gateway. Thereby, we avoided long component deployment delays. The CrimeSPOT components on the other hand, which have to be remotely deployed, have a minimal footprint. By taking the average of the sizes of our example applications' components, we ended up with an average component size of 1,99kB.

	SunSPOT App	LooCI Comp	CrimeSPOT Comp
RAM	43,28kB	119,57kB	131,96kB
Δ	43,28kB	76,29kB	12,39kB

Table B.2: Memory Footprint (RAM)

Table B.2 summarizes some RAM measurements. We verified the RAM usage by creating a SunSPOT application, a LooCI component, and a CrimeSPOT component, which all did nothing but outputting the RAM usage after a garbage collection. Since a CrimeSPOT component is actually a LooCI component, it comes as no surprise that running one with an empty Rule- and Fact Base doesn't introduce a lot of RAM overhead. A running CrimeSPOT system only consumes an additional 12,39kB of RAM when compared to a dummy LooCI component, bringing the total RAM usage to 131,96kB.

Since these RAM measurements only give an indication of the CrimeSPOT RAM usage at initialization time, we performed two additional experiments for measuring the RAM usage after adding to the Fact- and Rule Base, respectively.

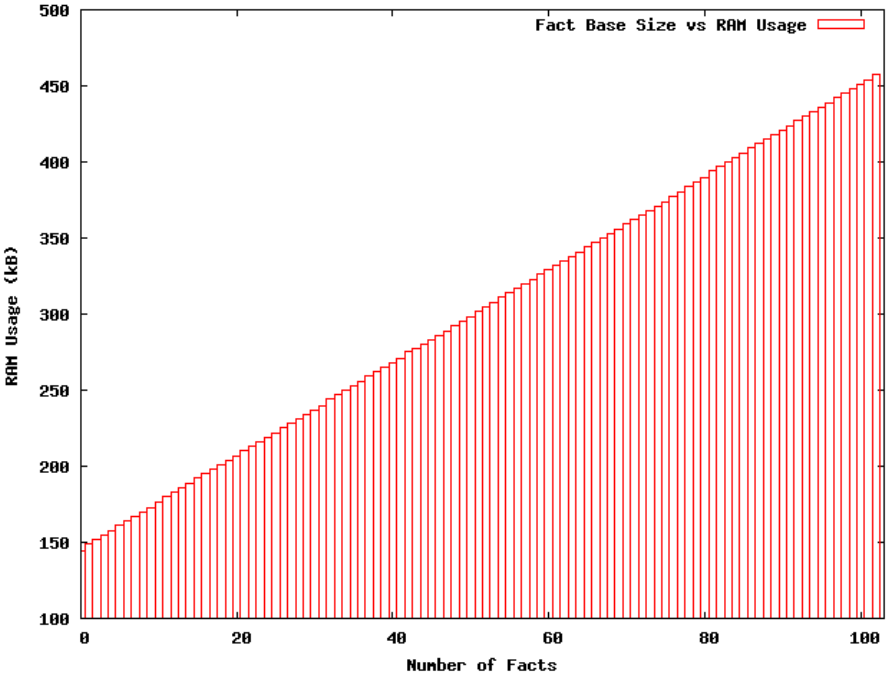


Figure B.1: Fact Base Size vs RAM Usage

Figure B.1 shows the RAM usage versus the amount of facts in the Fact Base. For this experiment, we asserted facts with two attributes each; one random Number between 0 and 50, and a constant Text value. This simulates a Fact Base filled with numerical sensor readings which have an additional description, a plausible scenario for CrimeSPOT components. In addition, we installed a rule with a matching condition that measured the RAM usage upon triggering. We kept asserting new facts until running out of memory. The

results are not very surprising: every asserted fact consumed an average of 3kB RAM and we were unable to assert more than 102 facts. However, these results clearly emphasize the importance of keeping the Fact Base clean (e.g., by using the subsumes and drop meta-facts).

	1 Rule	2 Rules	3 Rules	10 Rules
RAM	168,43kB	199,98kB	230,36kB	443,85kB

Table B.3: Memory Footprint (RAM) vs. Amount of rules

Table B.3 shows the RAM usage versus the amount of rules in the Rule Base. Since our example applications' rules have at most six conditions, we filled our Rule Base with distinct variants of the rule shown in Listing B.1 until running out of memory.

```
1 fact(A=?x, B=?y, C=?z)
2          <- c1(A=?x, B=?y, C=?z), c2(A=?x, B=?y, C=?z),
3             c3(A=?x, B=?y, C=?z), c4(A=?x, B=?y, C=?z),
4             c5(A=?x, B=?y, C=?z), c6(A=?x, B=?y, C=?z).
```

Listing B.1: Artificial rule for the RAM usage experiment

The artificial rule more or less represents a worst-case scenario in terms of memory consumption, as the body is quite large and the corresponding RETE network is therefore too. Furthermore, since all conditions are distinct, there's no reuse of filter nodes in the network. As shown in the results, the rule consumes around 30,5kB of RAM, and we were unable to instantiate it more than ten times. We attribute this considerable amount of memory consumption to the fact that the prototype implementation is not yet entirely cleaned up and optimized. Even though none of our example applications employed ten rules, and the average amount of conditions in their rules was smaller than six, the results of this experiment indicate considerable room for improvements, which are left for future work.

B.2 Network Overhead

A CRIMESPOT component doesn't introduce a lot of network overhead when compared to a regular LooCI component. Only two forms of network overhead can be identified: overhead in the publication of mapped facts, and overhead due to the publication of retraction events.

When publishing mapped facts, the predefined LooCI event payload is extended with the fact ID and -destination (i.e., the target MAC address and component ID). These extra fields are introduced for fact identification and unicast publication purposes, respectively. However, since the fact ID is only an integer, we consider this addition as a minimal overhead.

The remote retraction of published facts upon retracting them in the sender's Fact Base can also be observed as network overhead. However, the events published to retract published facts are restricted to a minimum. A retraction event doesn't carry the entire fact to be retracted, but only the data required for identifying that fact and its destination (i.e., the fact identifier and its destination MAC address and component ID). Furthermore, the publication of a retraction

event only occurs when the retracted fact has not yet expired, as this expiration would've also occurred at the CrimeSPOT components who received the fact.

B.3 Performance

To verify the performance of the CrimeSPOT prototype, we measured the time required for the work to be performed upon event reception. This section illustrates the computational overhead introduced by our prototype implementation.

When the received event entails the assertion of a fact, the event has to be transformed to a fact and the drop and subsumption meta-facts have to be verified before the fact can be asserted in the Fact Base. The average time required between receiving an event and asserting the associated fact is shown in Table B.4. For simplicity, we assumed the absence of matching drop and subsumption meta-facts. As the event transformation process is different for an

	Encoded fact	**Mapped fact**
Time (ms)	80,33	34,22

Table B.4: Average time required before a fact will be asserted in the Fact Base, after receiving the corresponding event

event containing an encoded fact (i.e., the fact's textual representation, e.g., `fact(Number=1, Text="a description")`), and a mapped event, we timed both. In this experiment, the event's corresponding fact to be asserted simulates a sensor reading or a typical application-fact; it has two attributes: a Number and a Text. We repeated the experiment ten times and ended up with an average delay of 80,33ms before asserting an encoded fact, and an average delay of 34,22ms before asserting a mapped fact. As expected, processing an event containing an encoded fact consumes more time as the complex textual representation has to be parsed before the corresponding fact can be asserted. The processing of mapped events is more efficient as a predefined event signature is typically small and the parsing is less complex (i.e., obtaining the fields from the event, and creating the corresponding fact suffices).

The presence of matching drop or subsumption meta-facts also introduces computational overhead. However, these meta-facts' bodies (i.e., the values of their When attribute) are almost completely verified *before* receiving a fact. When a newly received fact matches a meta-fact, the only work left to do is to join the token encapsulating this fact with the tokens for the meta-fact's body (i.e., in the join node that joins the Inference Layer's Rete network with the Reification Layer's network), which requires extra time linear in the amount of tokens for the body. In case there's a satisfied drop meta-fact, no further work is required and the fact is immediately dropped. In case there's a satisfied subsumption meta-fact, extra time will be required for retracting each subsumed fact.

To give an indication of the time required between the assertion of a fact and the activation of a rule, we re-used the artificial rule from Listing B.1. This rule is quite complex as it has six conditions and each condition shares variables

with all other conditions, requiring all these variables to be checked within the join nodes in the rule's RETE network. In the experiment, a matching fact was present for conditions c2 to c6, and we measured the time between the assertion of a matching c1 fact and the activation of the rule (i.e., the assertion of the fact from the rule's head). This simulates a worst-case scenario for this rule as the filter node for c1 is at the top of the RETE-network representing the rule, and the assertion of a matching c1 fact will thus introduce the most work (i.e., the token in which it is encapsulated will have to be propagated all the way from the top to the bottom of the RETE-network, passing every join node). We repeated this experiment ten times and ended up with an average delay of 141,20ms. *It must be noted that this average delay is only valid in this scenario. The real delay between asserting a matching fact and activating a rule depends on that rule's complexity, on the presence of matching facts for its other conditions, and on the amount of other rules with a condition matching the asserted fact preceding that rule in the Rule Base.*

When the received event entails the retraction of a fact, the corresponding fact has to be looked up in the Fact Base before it can be retracted, requiring constant time (i.e., an average of 28ms in our experiments).

To give an indication of the time required between the retraction of a fact and the deactivation of a rule, we performed about the same experiment as for the assertion of a fact. The rule from Listing B.1 was installed, a matching fact is present for all conditions, the rule has been activated, and we measured the time between retracting the matching c1 fact and the deactivation of the rule (i.e., the retraction of the fact from the consequence). For the same reasons as mentioned earlier, this experiment represents a worst-case scenario for this complex rule (i.e., the token has to propagate from the top to the bottom of the network). After repeating the experiment ten times, we ended up with an average delay of 188,90ms, which is surprisingly about 47ms higher than the delay introduced between asserting a matching c1 fact and activating the rule. Since fact retraction involves more or less the same processing as fact assertion, we attribute this difference to the fact that removing data from the data-structures employed within the RETE implementation, which is required upon fact retraction, is less efficient than adding data, which is required upon fact assertion. *For the same reasons as discussed earlier, it must be noted that this average delay between retracting the fact and deactivating the rule is only valid in this scenario.*

B.4 Conclusion

This appendix presented the preliminary evaluation of our CrimeSPOT prototype implementation. Since this implementation is not yet entirely cleaned up and optimized, we believe that the current results look quite promising. In the future, we expect to improve on these results by further reducing the memory footprint in terms of ROM and RAM requirements. Furthermore, additional optimizations and hardware advances could also improve the overall performance of our language.

Bibliography

[1] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, pp. 19:1–19:51, 2011.

[2] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Transactions on Sensor Networks*, vol. 4, no. 2, pp. 1–29, 2008.

[3] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[4] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha, "Actornet: an actor platform for wireless sensor networks," in *Proceedings of the 5th international joint conference on Autonomous Agents and MultiAgent Systems (AAMAS '06)*, pp. 1297–1300, 2006.

[5] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pp. 88–97, 2002.

[6] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet," in *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pp. 96–107, 2002.

[7] K. Martinez, J. K. Hart, and R. Ong, "Environmental sensor networks," *Computer*, vol. 37, no. 8, pp. 50–56, 2004.

[8] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita, "A line in the sand: a wireless sensor network for target detection, classification, and tracking," *Computer Networks*, vol. 46, no. 5, pp. 605–634, 2004.

[9] C. Hartung, R. Han, C. Seielstad, and S. Holbrook, "Firewxnet: a multitiered portable wireless system for monitoring weather conditions in wildland fire environments," in *Proceedings of the 4th international conference on Mobile Systems, Applications and Services (MobiSys '06)*, pp. 28–41, 2006.

[10] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. del Cid, S. Michiels, C. Huygens, W. Joosen, and J. Ueyama, "Building wireless sensor network applications with looci," *International Journal of Mobile Computing and Multimedia Communications*, vol. 2, no. 4, pp. 38–64, 2010.

[11] A. Deshpande, C. Guestrin, and S. R. Madden, "Resource-aware wireless sensor-actuator networks," *IEEE Data Engineering*, vol. 28, no. 1, 2005.

[12] I. Maier, T. Rompf, and M. Odersky, "Deprecating the observer pattern," tech. rep., Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2010.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, pp. 114–131, 2003.

[14] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 131–146, 2002.

[15] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[16] C. Srisathapornphat, C. Jaikaeo, and C. chung Shen, "Sensor information networking architecture and applications," *IEEE Personal Communications*, vol. 8, no. 4, pp. 52–59, 2001.

[17] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information Processing in Sensor Networks (IPSN '07)*, pp. 489–498, 2007.

[18] K. Whitehouse, J. Liu, and F. Zhao, "Semantic streams: a framework for composable inference of sensor data," in *Proceedings of the 3rd European Workshop on Wireless Sensor Networks (EWSN '06)*, pp. 5–20, 2006.

[19] S. Li, Y. Lin, S. H. Son, J. A. Stankovic, and Y. Wei, "Event detection services using data service middleware in distributed sensor networks," *Telecommunication Systems*, vol. 26, no. 2, pp. 351–368, 2004.

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[21] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java (tm) on the bare metal of wireless sensor devices: the squawk java virtual machine," in *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE '06)*, pp. 78–88, 2006.

[22] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[23] D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo, "A network-centric approach to embedded software for tiny devices," in *Proceedings of the 1st international workshop on Embedded Software (EMSOFT '01)*, pp. 114–130, 2001.

[24] L. Mottola and G. P. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *Proceedings of the 2nd international conference on Distributed Computing in Sensor Systems (DCOSS '06)* (P. Gibbons, T. Abdelzaher, J. Aspnes, and R. Rao, eds.), no. 4026 in Lecture Notes on Computer Science, (San Francisco (CA, USA)), pp. 150–167, Springer, June 2006.

[25] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, "Looci: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 7th international conference on Advances in Mobile Computing and Multimedia (MoMM '09)*, pp. 195–203, 2009.

[26] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, vol. 1, 2004.

[27] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Programming wireless sensor networks with the teenylime middleware," in *Proceedings of the ACM/IFIP/USENIX 2007 international conference on Middleware (Middleware '07)*, pp. 429–449, 2007.

[28] K. Terfloth, G. Wittenburg, and J. Schiller, "Facts - a rule-based middleware architecture for wireless sensor networks," in *Proceedings of the 1st international conference on COMmunication System softWAre and MiddlewaRE (COMSWARE '06)*, pp. 1–8, 2006.

[29] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[30] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," in *Proceedings of the international conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[31] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *Proceedings of the 5th international conference on Embedded networked Sensor Systems (SenSys '07)*, pp. 175–188, 2007.

[32] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: Design and implementation," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 690–699, 2004.

[33] Y. Ni, U. Kremer, A. Stere, and L. Iftode, "Programming ad-hoc networks of mobile and resource-constrained devices," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, pp. 249–260, 2005.

[34] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, 1997.

[35] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: a methodology for architecture-independent programming of networked sensor systems," in *Proceedings of the 2005 workshop on End-to-End, Sense-and-Respond systems, applications and services (EESR '05)*, pp. 19–24, 2005.

[36] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco, "Expressing sensor network interaction patterns using data-driven macroprogramming," in *Proceedings of the 5th IEEE international conference on Pervasive Computing and Communications Workshops (PERCOMW '07)*, pp. 255–260, 2007.

[37] C. Forgy, "Rete: A fast algorithm for the many pattern / many object pattern match problem," *Artificial Intelligence*, vol. 19, pp. 17–37, 1982.

[38] K.-U. Schmidt, R. Stühmer, and L. Stojanovic, "Blending complex event processing with the rete algorithm," in *iCEP2008: 1st international workshop on Complex Event Processing for the Future Internet colocated with the Future Internet Symposium (FIS '08)*, 2008.

[39] M. Perlin, "Scaffolding the rete network," in *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence (ICTAI '90)*, pp. 378–385, 1990.

[40] C. Scholliers and E. Philips, "Coordination in volatile networks," Master's thesis, Vrije Universiteit Brussel, 2007.