CRIMESPOT: a Language and Runtime for Developing Active Wireless Sensor Network Applications

Coen De Roover[a], Christophe Scholliers[a], Wouter Amerijckx[a], Theo D'Hondt[a], Wolfgang De Meuter[a]

[a]*Software Languages Lab, Vrije Universiteit Brussel, Belgium*

Abstract

Advances in wireless sensing and actuation technology allow embedding significant amounts of application logic inside wireless sensor networks. Such active WSN applications are more autonomous, but are significantly more complex to implement. Event-based middleware lends itself to implementing these applications. It offers developers fine-grained control over how an individual node interacts with the other nodes of the network. However, this control comes at the cost of event handlers which lack composability and violate software engineering principles such as separation of concerns. In this paper, we present CRIMESPOT as a domain-specific language for programing WSN applications on top of event-driven middleware. Its node-centric features enable programming a node's interactions through declarative rules rather than event handlers. Its network-centric features support reusing code within and among WSN applications. Unique to CRIMESPOT is its support for associating application-specific semantics with events that carry sensor readings. These preclude transposing existing approaches that address the shortcomings of event-based middleware to the domain of wireless sensor networks. We provide a comprehensive overview of the language and the implementation of its accompanying runtime. The latter comprises several extensions to the RETE forward chaining algorithm. We evaluate the expressiveness of the language and the overhead of its runtime using small, but representative active WSN applications.

Keywords: wireless sensor networks, ubiquitous computing, programming languages, software engineering

1. Introduction

Event-driven middleware enables the nodes of a wireless sensor network (WSN) application to communicate over a decentralized event bus. The middleware relieves developers from intricate concerns such as resource management and volatile connections. Even with these concerns out of the way, a node's communication with other nodes is often difficult to program. Different events need to be reacted to differently at run-time. This usually implies some form of dispatching over each event that is received. Reacting to a sequence of events implies keeping track of how many events of the sequence have already occurred. Without adequate middleware support for these problems, developers have to resort to ad-hoc solutions in event handlers. However, such solutions have been shown to violate a range of software engineering principles [18]. A node's event handler, for instance, cannot simply be composed with another to have it react to an additional event sequence. While these problems plague other event-driven architectures as well, existing solutions do not readily translate to WSNs. Next to messages concerned with application logic, events carry sensor readings that have to be handled as such. Small fluctuations in the payload of successive events might not warrant a reaction. When a node disconnects, on the other hand, some of the state changes it induced in other nodes might have to be undone as well. Not only will ad-hoc solutions lead to code duplication, they will also require a considerable amount of bookkeeping.

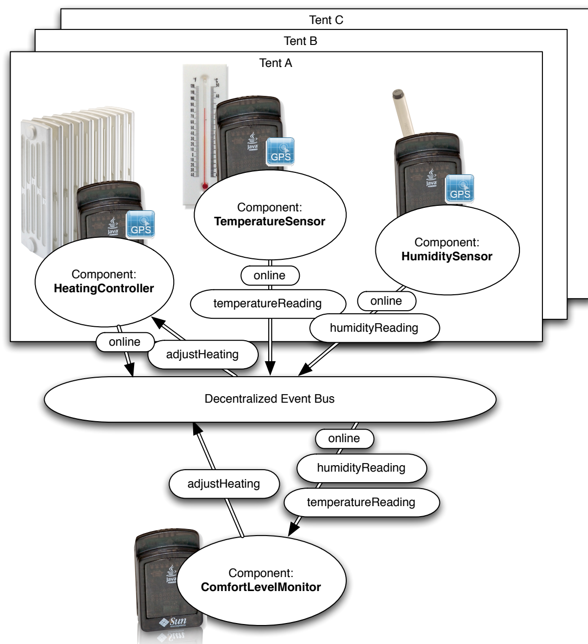

Figure 1: Motivating example: a WSN application for festival tents.

FR1 The *HumiditySensor* and *TemperatureSensor* have to publish their sensor readings at set intervals.

FR2 The *HumiditySensor*, *TemperatureSensor* and *HeatingController* have to publish their online presence and location at set intervals.

FR3 The *HeatingController* has to adjust its associated heater according to a received `adjustHeating` event.

FR4 The *ComfortLevelMonitor* has to compute a tent's heating level based on a received `temperatureReading` event and publish this level in an `adjustHeating` event.

FR5 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.

FR6 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.

FR7 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.

FR8 The *HeatingController* has to make sure that its associated heater won't keep heating when the *ComfortLevelMonitor* fails or gets disconnected from the WSN.

Figure 2: Functional requirements for the example.

2. Motivating Example

Figure 1 further illustrates the problems identified above. An event-based WSN application has been deployed to control the heaters in several festival tents. A *HeatingController*, *TemperatureSensor* and *HumiditySensor* node is deployed in each tent. As a malevolent attendee might bring these nodes offline or move them between tents, they are fitted with a GPS sensor that is used to communicate their location (**FR2**). All nodes communicate over a decentralized event bus. Outgoing arrows depict events published by a node, while incoming arrows depict events a node is subscribed to. Each *TemperatureSensor* and *HumiditySensor* node continuously publishes `temperatureReading` and `humidityReading` events on the event bus. *HeatingController* nodes subscribe to *adjustHeating* events. Upon receiving an `adjustHeating` event, they adjust the setting of the heater they are associated with. Such events are published by the *ComfortLevelMonitor* which decides when and by how much each heater needs to be adjusted based on the `temperatureReading` events it receives. This node also logs the comfort levels in each festival tent over time. To this end, it combines the data carried by `temperatureReading` events with those carried by `humidityReading` events. Note that a single *ComfortLevelMonitor* node monitors and controls the comfort levels in all tents. Care must therefore be taken not to combine `temperatureReading` and `humidityReading` events that originate from different tents. Figure 2 summarizes the functional requirements for our motivating example.

2.1. Reacting to Events using Event Handlers

The first three requirements amount to *invoking application logic* or *publishing a new event* whenever a node receives an event. Most event-based middleware supports implementing such reactions in a node's event handler (e.g., a method `receiveEvent(Event)` invoked by the middleware). To implement (**FR3**), for instance, the event handler of *HeatingController* merely has to read out the payload of each received `adjustHeating` event and adjust its heater accordingly. No other reactions to such an event are required, nor are there any other events the node is subscribed to.

The event handler of the *ComfortLevelMonitor*, in contrast, has to *dispatch* over `online`, `humidityReading` and `temperatureReading` events of which the latter requires multiple reactions. Not only must an `adjustHeating` event be

Email addresses: `cderoove@vub.ac.be` (Coen De Roover), `cfscholl@vub.ac.be` (Christophe Scholliers), `wamerijc@vub.ac.be` (Wouter Amerijckx), `tjdhondt@vub.ac.be` (Theo D'Hondt), `wdmeuter@vub.ac.be` (Wolfgang De Meuter)

published (**FR4**), but a comfort level has to be computed as well from the `temperatureReading` and a previously or yet to be received `humidityReading` (**FR6**). This typically entails *storing* received events in memory such that they can be consulted and related with other events later on. Relating events usually involves *matching* their payloads and/or information about their origin. For instance, the payload of `online` events relates node identifiers with tent identifiers (i.e., which node resides in which tent). Stored `online` events can therefore be used to determine which tent an event originated from. This is necessary to ensure that comfort levels are computed using events that originate from the same tent (**FR6**).

Without adequate language or middleware support for the aforementioned event *dispatching*, *storage* and *matching*, developers have to resort to ad-hoc implementations. These are error-prone and bound to be duplicated over the event handlers of multiple nodes. Furthermore, a node's event handler cannot easily be composed with another to have it react to an additional event.

2.2. Semantics of Events that Carry Sensor Readings

While the above problems plague other event-driven applications as well, existing solutions (e.g., complex event processing) do not readily translate to WSNs. The semantics of events that carry sensor readings differs significantly from those that are intended to steer application logic.

First of all, one can wonder how long a received event remains valid (i.e., still warrants being reacted to later on). The `temperatureReading` and `humidityReading` events might not be published at the same interval. Storing either until the corresponding event is received, might lead to comfort levels being computed from stale information (**FR4**). One might therefore want to associate an *expiration* time with events that carry sensor readings —in contrast to events that are concerned with distributed application logic.

Furthermore, multiple temperature sensors can be deployed in the same tent. Small fluctuations in the payload of successive `temperatureReading` events might therefore not warrant a reaction (**FR3**) every time one is received. The comfort levels logged by *ComfortLevelMonitor* should, on the other hand, always be computed from the most recently received sensor readings (**FR7**). Likewise, a newly received `online` event immediately invalidates the information carried by older ones. All of these issues concern the *subsumption* of an older event by a newer one.

Finally, previous reactions to expired or subsumed events might even have to be *compensated* for. For instance, the *HeatingController* should reset its associated heater if no `adjustHeating` event has been received for some time. This way, it can avoid overheating a tent when the *ComfortLevelMonitor* fails or gets disconnected (**FR8**). In general, compensating for expired events entails tracking the causality between events and the reactions they triggered over the WSN.

Even with the aforementioned event *dispatching*, *storage* and *matching* supported by the middleware or programming language, implementing event *expiration*, *subsumption* and *compensation* still involves a fair amount of bookkeeping. In this paper, we present CrimeSPOT as a language that is explicitly designed to minimize the accidental complexity that is inherent to programming WSN applications using event-based middleware. CrimeSPOT enables developers to focus on the application's essential complexity instead.

3. Overview of the Approach

CrimeSPOT is a domain-specific programming language to be used on top of event-based middleware for wireless sensor networks. From the *node-centric perspective*, it enables programming the interactions of a node with other nodes on the network through declarative rules rather than event handlers. Each rule specifies how the node should react to a particular sequence of events. This way, developers are relieved from having to dispatch explicitly over each received event and having to track how many events of an event sequence have already been received. More importantly, interactions can be composed by enumerating the rules that govern them. From the *network-centric perspective*, CrimeSPOT enables developers to specify which rules are to govern which nodes of the network. Through macro-programming facilities, the resulting configurations of nodes and rules can be reused across WSN applications.

Tailored towards WSNs, CrimeSPOT explicitly supports associating application-specific semantics to events that carry sensor readings. This includes determining which network events correspond to a sensor reading, but also when a sensor reading expires, when a sensor reading subsumes a previous one and when and how often a new reading warrants triggering an interaction rule. In addition, CrimeSPOT tracks causality relations between the events a node

receives and the ones it publishes. This allows determining whether, which and how nodes are affected when a sensor reading is subsumed or expires.

Figure 5 and Figure 6 depict the CrimeSPOT implementation of the motivating example. Without delving into details, the interaction rule on lines 7–9 of Figure 5 specifies that a `temperatureReading` should be published to all network nodes periodically. Section 4 and Section 5 discuss the CrimeSPOT features it relies on from the node-centric and network-centric perspective respectively. We will discuss its accompanying runtime first.

3.1. *Architecture of the* CrimeSPOT *Runtime*

An instance of the CrimeSPOT runtime has to be instantiated on every network node of which the interactions are to be governed by CrimeSPOT rules. Figure 3 depicts the layered architecture of this runtime. The *middleware bridge* in the infrastructure layer binds the runtime to the underlying event-based middleware. It contains middleware-specific functionality to transfer events from and to the decentralized event bus.

The *reification engine* in the reification layer reifies the events that are received from other nodes as facts and stores them in a *fact base*. This enables the natural use of pattern matching in rules to relate stored events through their payload or origin. Section 3.3 discusses how the reification engine can be tailored to the specifics of a WSN application by storing declarations in its *configuration base*. Among others, an expiration time can be associated with a fact that reifies an event.

Next to the aforementioned fact base, the inference layer contains a *rule base*. As soon as an interaction rule has been added to the rule base, it intervenes in how the node processes the events received on the event bus. To this end, the *inference engine* re-evaluates the fact base against the rule base whenever the former changes —at least, conceptually. Section 3.2 discusses how the inference engine evaluates interaction rules incrementally.

Interaction rules consist of a body and a head separated by the neck symbol "`<-`" (cf. lines 7–9 of Figure 5). In general, the body of a rule consists of conditions that correspond to events that have been received and stored as facts. They therefore express which events must have been received in order for the rule to be *activated*. The head of most rules consists of a fact. Whenever such a rule is activated, the inference engine adds the fact in its head to a fact base. Meta-data (i.e., everything between `@[...]` such as the `to(MAC=*)` on line 7) determines whether the fact is added to the fact base of the local node or to those of the other nodes on the network. Application logic can also be invoked when a rule is activated. The head of such rules consists of a reference to a field (e.g., `this.adjustHeater` on line 29 of Figure 5), the value of which will be sent a message `activated(CSVariableBindings)` upon rule activation. The corresponding method can be used to implement application logic (e.g., adjust heater).

Note that an activated rule can become *deactivated* in a successive evaluation of the rule base against the fact base. This is the case as soon as one of its conditions is no longer satisfied. For instance, because the matching fact expired and was removed from the fact base. The inference engine will undo all reactions to a rule's activation upon its deactivation. For rules with a fact in their head, this fact will be removed from all fact bases it was added to. For rules with a field reference in their head, a message `deactivated(CSVariableBinding)` will be sent to the value of the field upon their deactivation. The corresponding method can be used to implement compensating application logic (e.g., reset heater). Section 3.2 discusses how the inference engine tracks the causality between rule bodies and heads.

3.2. *Inference Engine of the* CrimeSPOT *Runtime*

The inference engine of the CrimeSPOT runtime evaluates the rule base against the fact base whenever the latter changes. To this end, the engine uses forward chaining as its inference strategy. Working from the body of a rule to its head, forward chaining derives all conclusions that follow from a fact base. Backward chaining, in contrast, gathers facts supporting a given conclusion —working from the head of a rule to its body. Backward chaining is goal-driven whereas forward chaining is data-driven. The latter lends itself to an incremental evaluation of the rule base against the fact base. Incremental evaluation is essential in our setting, as the fact base is updated frequently (e.g., whenever an event is received).

The Rete algorithm [9] is an incremental forward chainer that sacrifices memory for speed. The algorithm stores intermediate derivations and combines them with a newly added fact to derive the additional conclusions that follow from the augmented fact base. This way, not all conclusions have to be re-derived from scratch whenever a fact is added to the fact base. In past work, we extended the Rete algorithm into the distributed truth maintenance system Crime [22]. Crime explicitly tracks the causal links between facts and conclusions, including distributed ones. This allows computing the conclusions that no longer follow from a reduced fact base. Being able to react to such invalidated

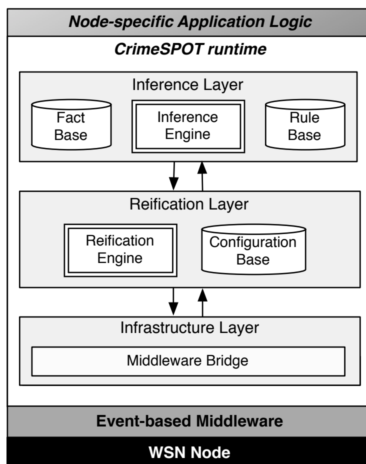

Figure 3: Architectural overview of the CrimeSPOT runtime.

conclusions is vital to the way CrimeSPOT reconciles the transient nature of events with the persistent nature of facts; through customizable event reification and fact expiration. These features require CrimeSPOT-specific extensions to the Rete algorithm, which are detailed in Section 7.

3.3. *Reification Engine of the* CrimeSPOT *Runtime*

The reification engine of the CrimeSPOT runtime reifies transient events as persistent facts. As mentioned before, its behavior can be tailored completely to the specifics of a WSN application through declarations. These declarations control which and how events are to be reified as facts, when the resulting facts expire and which older facts are subsumed by a new fact.

Figure 4 illustrates the reification process. Each incoming event is reified as a fact first. If the event wraps a fact, reifying the event is trivial (i.e., the fact has to be unwrapped from the event). This is the case for facts that have been published through CrimeSPOT rules. Otherwise, the incoming event must have originated from a node that does not run the CrimeSPOT runtime on top of the WSN middleware. This is typically the case for resource-constrained nodes that only publish events with sensor information. When such an event is received, the reification engine consults declarations of the form "`mapping <fact> <=> <event>`". Lines 39–40 of Figure 6 depict an example of such a declaration.[1] It specifies that a middleware event of type `101` with a single `Integer` payload is to be reified as a `temperatureReading` fact with a single attribute named `Celsius`. The occurrences of variable `?temp` ensure that the value of the fact's attribute corresponds to the payload of the event.

Next, the reification engine consults the declarations of the form "`drop <fact> [provided <conditions>]`". These determine whether the newly created fact should be added to the fact base. This might not be the case if existing facts subsume the newly created fact. If the fact doesn't have to be dropped, the engine consults the declarations of the form "`incoming <newfact> subsumes <oldfact> [provided <conditions>]`". These determine which facts have to be removed from the fact base because they are subsumed by the newly created fact. Only then, the new fact is added to the fact base.

Consider the declaration on line 27 of Figure 5. It specifies that a new `adjustHeatingLevel` fact subsumes all other facts of the same type. As a result, the fact base of the *HeatingController* node will always contain the most recently received fact. The declaration on lines 5–6 of Figure 6 specifies that an `online` fact subsumes other `online` facts that were received from the same network node. To this end, variable `?m` substitutes for the MAC-address of the node that published the new fact (in the `from` meta-data on line 5) as well as for the MAC-address of the node that published the older fact (in the `from` meta-data on line 6). Note that a different variable substitutes for the value of the `Tent`-attribute of both facts. Indeed, nothing precludes a node from being moved.

[1]Note that this particular declaration could be omitted from the motivating example as `temperatureReading` facts are already published by a CrimeSPOT node (i.e., lines 7–9 of Figure 5).

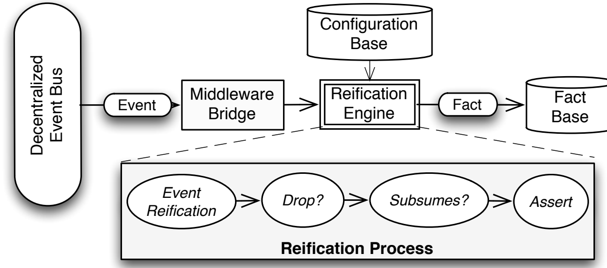

Figure 4: The reification engine of the CrimeSPOT runtime.

```
1.    TemperatureSensor, HumiditySensor, HeatingController {
2.        publishPresenceEvery($onlineInterval).
3.    }
4.
5.    TemperatureSensor {
6.        temperatureMapping($readingInterval).
7.        temperatureReading(Celsius=?temp)@[to(MAC=*),
8.                                           factExpires($readingInterval)]
9.          <- ?temp is this.getTemperature()@[renewEvery($readingInterval)].
10.   }
11.
12.   TemperatureSensor.java {
13.       private CSValue getTemperature() { return ... }
14.   }
15.
16.   HumiditySensor {
17.       humidityMapping($readingInterval).
18.       humidityReading(Percent=?p)@[to(MAC=*),
19.                                    factExpires($readingInterval)]
20.         <- ?p is this.getHumidity()@[renewEvery($readingInterval)]
21.   }
22.   HumiditySensor.java {
23.       private CSValue getHumidity() { return ... }
24.   }
25.
26.   HeatingController {
27.       incoming adjustHeating(Level=?new) subsumes adjustHeating(Level=?old).
28.
29.       this.adjustHeater
30.         <- adjustHeating(Level=?h).
31.   }
32.
33.   HeatingController.java {
34.     private CSAction adjustHeater = new CSAction() {
35.        public void activated(CSVariableBindings bindings) { //adjust heating }
36.        public void deactivate(CSVariableBindings bindings) { //reset heating }
37.     };
38.   }
```

Figure 5: CrimeSPOT code for the *TemperatureSensor*, *HumiditySensor* and *HeatingController* nodes in the motivating example.

4. Node-centric CrimeSPOT Features

The preceding sections introduced the runtime that supports the CrimeSPOT language. Here, we introduce the node-centric features of this programming language. These enable programming the interactions of a node with others through declarative rules rather than the predominant event handlers. Figure 7 depicts the grammar of all node-centric CrimeSPOT features. We will focus on the features that are required to understand the CrimeSPOT implementation of the motivating example. First, we discuss the activation and deactivation of rules in more detail.

4.1. Rule Activation and Deactivation

The body of a CrimeSPOT rule corresponds to a conjunction of conditions. Each condition has to be satisfied in order for the rule to be activated. A condition is satisfied if a matching fact exists in the node's fact base. Facts consist of a functor, a sequence of named attributes and optional meta-data. The `=` symbol separates the name of each attribute from its value. Meta-data is demarcated by a `@[...]` construct. Among others, the fact-like declarations within this construct record information about the fact's origin. For instance, the fact base of the *ComfortLevelMonitor* node contains facts of the following form:

```
temperatureReading(Celsius=27)
 @[factExpires(Seconds=600),from(MAC=1234:1234:1234:1234)]
```

The syntax for conditions is similar to the one of facts, except that a logic variable (i.e., an identifier starting with a question mark) can substitute for the concrete value of a named attribute. For a condition to be satisfied, there has to be a fact that matches the condition under a variable substitution (i.e., a mapping of variables to the values they are bound to). Note that a fact can match a condition with less attributes and meta-data. The fact only has to exhibit the attributes and meta-data that are specified in the condition. This is why CrimeSPOT uses named attributes.

The bindings for each occurrence of a variable have to be consistent across the head and the body of a rule. One rule activation therefore corresponds to a particular substitution for its variables. If the fact base contains three matching `temperatureReading` facts, for instance, the rule is activated three times with a corresponding binding for `?temp`:

```
1.    ComfortLevelMonitor {
2.        temperatureMapping($readingInterval).
3.        humidityMapping($readingInterval).
4.
5.        incoming online(Tent=?tnt,Node=?n)@[from(MAC=?m)]
6.        subsumes online(Tent=?otnt,Node=?n)@[from(MAC=?m)].
7.
8.        subsumesOlderFromSameTent(humidityReading,Percent).
9.        subsumesOlderFromSameTent(temperatureReading,Celsius).
10.
11.       this.logComfortLevel
12.         <- humidityReading(Percent=?h)@[from(MAC=?hm)],
13.            online(Tent=?tnt)@[from(MAC=?hm)],
14.            temperatureReading(Celsius=?t)@[from(MAC=?tm)],
15.            online(Tent=?tnt)@[from(MAC=?tm)].
16.
17.       adjustHeating(Level=?heatingLevel)@[to(MAC=?hcm),
18.                                           factExpires($readingInterval)]
19.         <- temperatureReading(Celsius=?t)@[from(MAC=?tm)],
20.            online(Tent=?tnt)@[from(MAC=?tm)],
21.            ?heatingLevel is this.computeHeatingLevel((Number)?t),
22.            online(Node=HeatingController,Tent=?tnt)@[from(MAC=?hcm)].
23.   }
24.
25.   ComfortLevelMonitor.java {
26.      private CSValue computeHeatingLevel(Number t) { return ... }
27.      private CSACtion logComfortLevel = new CSAction() { ... }
28.   }
```

```
29.   *.java {
30.       private CSValue getTentBasedOnGPSReading() { return ... }
31.   }
32.
33.   * {
34.
35.       defvar $readingInterval: Seconds=600.
36.       defvar $onlineInterval: Seconds=3600.
37.
38.       defmacro temperatureMapping():
39.           mapping temperatureReading(Celsius=?temp)@[factExpires($readingInterval)]
40.              <=> Event_101(Integer=?temp).
41.
42.       defmacro humidityMapping():
42.           mapping humidityReading(Percent=?h)@[factExpires($readingInterval)]
44.              <=> Event_102(Integer=?h).
45.
46.       defmacro publishPresenceEvery($time):
47.          online(Tent=?tnt,Node=$NAME)@[to(MAC=*),factExpires($time)]
48.             <- ?tnt is this.getTentBasedOnGPSReading()@[renewEvery($time)].
49.
50.       defmacro subsumesOlderFromSameTent($reading,$type):
51.          incoming $reading($type=?new)@[from(MAC=?mac)]
52.          subsumes $reading($type=?old)@[from(MAC=?othermac)]
53.          provided online(Tent=?tnt)@[from(MAC=?mac)],
54.                   online(Tent=?tnt)@[from(MAC=?othermac)].
55.   }
```

Figure 6: CrimeSPOT code for the *ComfortLevelMonitor* (left) and code that is shared by all nodes in the motivating example (right).

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

The fact base will therefore be extended with three new `temperature` facts. As soon as a new `temperatureReading` fact is added to fact base, the rule is activated anew with another variable substitution that results in a new `temperature` fact. Conversely, as soon as a `temperatureReading` fact is removed from the fact base, the rule will be deactivated for the corresponding variable substitution. As a result, one of the `temperature` facts produced by this rule will be removed.

4.2. Relating Facts

As illustrated by the motivating example, WSN nodes often have to store and relate the events they receive. Through multiple occurrences of a variable in a rule's body, CrimeSPOT supports relating facts that reify received events based on their content as well as their origin. Consider the interaction rule of the *ComfortLevelMonitor* on lines 11-15 of Figure 6. The first two conditions succeed if both a `humidityReading` and an `online` fact are stored in the fact base. However, variable `?hm` requires these facts to have originated from the same network node. Within the meta-data of each condition, the variable substitutes for the MAC address the fact was published from. As a result, variable `?tnt` will be bound to the tent from which the `humidityReading` originated. This is an example of *origin-based relating of facts*. The last two conditions use another occurrence of this variable to find a `temperatureReading` from the same tent. This is an example of *content-based relating* of facts.

The same technique can be used to link the head of a rule to its body. This is illustrated by the rule on lines 17–22 of Figure 6. The occurrences of `?hcm` ensure that an `adjustHeatingLevel` fact is added to the fact base of the particular heating controller in the tent from which the temperature reading originated. By default, facts are only added to the local fact base. This behavior is changed by the `to(MAC=?hcm)` declaration in the fact's meta-data. Likewise, a `to(MAC=*)` declaration will add the fact to all fact bases. If the underlying middleware does not support such unicasts, the infrastructure layer of our runtime will simulate them through broad-casts that are filtered at the receiver side.

Both rules have multiple conditions in their body. Note that there merely has to be a matching fact in the fact base for each condition. A rule does not by itself specify an order in which the corresponding events must have been received. This is appropriate as sensor readings arrive non-deterministically. The aforementioned subsumption declarations ensure that only the most recent sensor readings are stored for each tent. Otherwise, the first rule would be activated multiple times: once for each combination of humidity and temperature readings that are stored. In general, an interaction rule cannot be understood in isolation from the declarations that configure the reification engine.

4.3. Error Handling

Some form of error handling might be in order when a match cannot be found for a condition. To this end, a `matchEvery` declaration can be added to the meta-data of the condition. Whenever a match hasn't been found in the specified amount of time, a `timedOut` fact will be added to the local fact base. The condition in the following rule expects a new matching fact at least every minute:

```
⟨node-centric⟩   ::= ⟨fact⟩ | ⟨rule⟩ | ⟨declaration⟩
⟨fact⟩           ::= ⟨identifier⟩([⟨attribute⟩(,⟨attribute⟩)*])[@[⟨meta-datum⟩(,⟨meta-datum⟩)*]]
⟨attribute⟩      ::= ⟨identifier⟩⟨relop⟩(⟨value⟩|⟨variable⟩)
⟨relop⟩          ::= = | != | <= | >= | < | >
⟨variable⟩       ::= ?⟨identifier⟩
⟨value⟩          ::= ⟨list⟩ | ⟨boolean⟩ | ⟨number⟩ | ⟨string⟩ | ⟨mac-address⟩
⟨val-or-var⟩     ::= ⟨value⟩|⟨variable⟩
⟨list⟩           ::= [] | [⟨val-or-var⟩(,⟨val-or-var⟩)*]
⟨meta-datum⟩     ::= to(MAC=(⟨mac-address⟩|⟨variable⟩)|*)
                  |  factExpires(Seconds=⟨number⟩)
                  |  from(MAC=(⟨mac-address⟩|⟨variable⟩)|this.MAC)
                  |  matchExpires(Seconds=⟨number⟩)
                  |  matchEvery(Seconds=⟨number⟩)
⟨rule⟩           ::= ⟨head⟩ <- ⟨body⟩
⟨head⟩           ::= ⟨fact⟩ | ⟨reference⟩
⟨reference⟩      ::= this.⟨identifier⟩
⟨body⟩           ::= ⟨condition⟩(,⟨condition⟩)*
⟨condition⟩      ::= ⟨fact⟩ | ⟨extra-logical⟩
⟨extra-logical⟩  ::= ⟨val-or-var⟩ is ⟨invocation⟩[@[⟨inv-option⟩]]
                  |  findall(⟨val-or-var⟩,[⟨body⟩],⟨val-or-var⟩)
                  |  length(⟨val-or-var⟩,⟨val-or-var⟩)
                  |  not ⟨condition⟩
                  |  ⟨val-or-var⟩⟨relop⟩⟨val-or-var⟩
⟨invocation⟩     ::= this.⟨identifier⟩([⟨val-or-var⟩(,⟨val-or-var⟩)*])
⟨inv-option⟩     ::= evalEvery(Seconds=⟨number⟩)
                  |  renewEvery(Seconds=⟨number⟩)
⟨declaration⟩    ::= mapping ⟨fact⟩ <=> ⟨fact⟩
                  |  incoming ⟨fact⟩ subsumes ⟨fact⟩ [provided ⟨body⟩]
                  |  drop ⟨fact⟩ [provided ⟨body⟩]
```

Figure 7: Grammar of node-centric CrimeSPOT.

```
⟨network-centric⟩        ::= (⟨quantified-block⟩)*
⟨quantified-block⟩       ::= ⟨crimespot-quantifier⟩ { (⟨crimespot-code⟩. )* }
                          |  ⟨middleware-quantifier⟩ { (⟨middleware-code⟩)* }
⟨middleware-quantifier⟩  ::= *.java | ⟨identifier⟩.java (, ⟨identifier⟩.java)*
⟨crimespot-quantifier⟩   ::= * | ⟨identifier⟩ (, ⟨identifier⟩)*
⟨crimespot-code⟩         ::= ⟨node-centric⟩
                          |  defvar ⟨macro-var⟩:⟨macro-val-or-var⟩
                          |  defmacro ⟨identifier⟩(⟨macro-var⟩(,⟨macro-var⟩)*)
                          |  ⟨identifier⟩(⟨macro-val-or-var⟩(,⟨macro-val-or-var⟩)*)
                          |  import ⟨path-to-file⟩
⟨macro-var⟩              ::= $⟨identifier⟩
⟨macro-val-or-var⟩       ::= ⟨macro-var⟩ | ⟨text⟩
```

Figure 8: Grammar of network-centric CrimeSPOT.

```
gotReadingFrom(MAC=?mac)
  <- temperatureReading(Celsius=?)@[from(MAC=?mac),
                                    matchEvery(Seconds=60)].
```

Whenever such a fact has not arrived one minute after the last one, the following `timedOut` fact will be asserted:

```
timedOut(Head=gotReadingFrom_1,Condition=temperatureReading).
```

The inference engine will activate the error handling rule with the corresponding `timedOut` condition in its body. As soon as a match is found for the condition that timed out, the `timedOut` fact will be removed from the fact base. Consequentially, the error handling rule will be deactivated as well.

4.4. Match Expiration

CrimeSPOT supports a `matchExpires` declaration among the meta-data of a condition. Any fact that matches such a condition will only match the condition for the specified amount of time. The following condition therefore only matches temperature readings that have been in the fact base for no more than 10 seconds:

```
temperatureReading(Celsius=?temp)@[matchExpires(Seconds=10)]
```

Note that the expiration of the match does not imply the expiration of the matched fact.

4.5. Invoking Application Logic

Application logic can be invoked from within the body or from the head of an interaction rule. Although other ports are possible, our run-time currently expects the underlying middleware to be executed on the Squawk VM [26]. A node's application logic therefore has to be implemented in this Java variant. The next section will discuss network-centric features of CrimeSPOT that enable specifying interaction rules and application logic in a uniform manner.

As discussed before, rules can have a reference to a field in their head (e.g., `this.adjustHeater` on line 29 of Figure 5). When such a rule is activated, the inference engine sends a message `activated(CSVariableBindings)` to the value of this field. The bindings for the variables in the rule's body are given as an argument. The corresponding Java method is to implement the application logic. Conversely, the message `deactivated(CSVariableBinding)` is sent to the value of the field upon the rule's deactivation. The corresponding method can compensate for the other. This is

particularly useful for error handling rules that were activated because of timeouts. Among others, state changes can be undone.

Java methods can also be invoked from within the body of a rule. To this end, CrimeSPOT supports conditions of the form "`<variable> is <invocation>`". Such a condition binds the variable on its left-hand side to the result of the invocation on the right-hand side. Usually, is-conditions have either a `renewEvery` or an `evalEvery` declaration among their meta-data. Both schedule the method to be invoked at set intervals. The former declaration invalidates previous matches for the condition, thus causing a deactivation of the rule in which it resides. The latter declaration gives rise to multiple matches for the is-condition, each with a different binding for the variable on the left-hand side. This can be useful to store a log of sensor readings in a node's fact base, but is memory-intensive.

The rule on lines 7–9 of Figure 5 uses an is-condition with a `renewEvery` declaration. As a result, method `getTemperature` is invoked periodically. Note that the `temperatureReading` facts published by this rule are declared to expire after the same interval. This is an optimization that allows the *TemperatureSensor* node to forego ordering all other nodes to remove a `temperatureReading` every time the rule is deactivated. Instead, the fact will have been removed already because it expired.

4.6. Negation, Aggregation and Relational Operators

As our inference engine extends Crime [22] with domain-specific features, CrimeSPOT inherits the latter's support for negation (i.e., `not`), aggregation (i.e., `findall`) and relational operators (e.g, `>`). For instance, the following condition has a match (without any variable bindings) for as long as the fact base contains no `adjustHeating` fact:

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

Relational operators are supported as well. The following conditions have a match for as long as there are at least two temperature readings such that one reading is higher than the other:

```
temperatureReading(Celsius=?t1),
temperatureReading(Celsius=?t2),
?t1 > ?t2
```

The above relational operator was used as a condition on its own. Relational operators can also be used within a condition to express a relational on an attribute of a matching fact. For instance, if the value of the second `Celsius` attribute is irrelevant, the above condition can be rewritten as follows

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

Finally, the `findall` operator can be used to gather values in a list. CrimeSPOT demarcates lists by square brackets [and]. The `findall` operator takes three arguments. At any time, the operator has a match that binds its last argument to a list that corresponds to the values of its first argument (i.e., a variable) across all matches for its second argument (i.e., a list of conditions). Conceptually, these conditions are re-evaluated whenever the fact base changes. The following conditions have a match for as long as at least three `smoke` facts exist in the fact base:

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

5. The Network-centric CrimeSPOT Features

Having discussed the node-centric features of the CrimeSPOT programming language, we shift our focus to its network-centric features. Their grammar is depicted in Figure 8. Used to specify which rules and application logic are to govern the behavior of which nodes, these features provide a holistic view of the WSN application as a whole. In this view, it is immediately apparent which nodes communicate with each other and how often. Macro definition and application provides an indispensable means to abstract and reuse code within and among WSN applications.

The resulting network-centric applications, such as the one depicted in Figure 5 and Figure 6, are compiled into node-level code that is tailored to each individual WSN node. The underlying event-driven middleware is the compilation target. Section 6 discusses the compilation process in more detail. The resulting code can be deployed as is through the middleware's over-the-air deployment facilities. It includes a CrimeSPOT runtime of which the rule base and configuration base have been populated.

5.1. Quantified Code Blocks

A CrimeSPOT file consists of blocks of code for each node required by the WSN application. Two kinds of blocks can be distinguished. The first kind groups node-centric CrimeSPOT code such as interaction rules and the declarations that configure the reification engine. These are demarcated by braces preceded by a quantifier. This quantifier specifies the WSN node for which the code is intended. For instance, lines 5–10 of Figure 5 group all the code for the *TemperatureSensor* node.

The second kind of blocks groups code that implements application logic in the language supported by the underlying middleware. They are similar to the other blocks, except that their quantifiers are suffixed with `.java`. Our prototype expects the underlying middleware to be executed on the Squawk VM [26]. Application logic therefore has to be implemented in this Java variant. For instance, lines 12–14 of Figure 5 group all the application logic required by the *TemperatureSensor* node.

When a code block is to be shared by multiple WSN nodes, it suffices to use an enumeration of their names as the quantifier. This is illustrated by the first line of Figure 5. Furthermore, a *-wildcard can be used for blocks that are to be shared by all WSN nodes. This is illustrated by the blocks on the right-hand side of Figure 6.

5.2. Macros and Macro Variables

CrimeSPOT supports macro variables within code blocks. Such variables are prefixed by a `$`-sign and are either predefined or defined by the developer within the scope of a particular code block. Macro variables substitute for a textual value at compile-time. They do not exist anymore at run-time. The predefined macro variable `$NAME` can be used wherever the name of a node is expected. This is useful when quantifying over multiple nodes. Line 47 of Figure 6 uses this macro variable to pass the name of a node as an attribute of the online facts it publishes. Line 35 of Figure 6 has the `$readingInterval` macro variable substitute for the `Seconds=600` attribute. To ensure all sensor nodes publish their readings at the same interval, this variable is defined in the scope of a block that is quantified by the *-wildcard. Among others, the variable is referred to by the *TemperatureSensor* on lines 7–9 of Figure 5. Used in this manner, macro variables ensure that a WSN application is easier to reconfigure.

Procedure-like macros can also be defined. For instance, lines 50–54 of Figure 6 define the macro `subsumesOlderFromSameTent($,$)`. It substitutes for a subsumption declaration specifying that a fact of type `$reading` with an attribute named `$type` subsumes all older `$reading` facts that are received from a node in the same tent. This macro is applied with the required arguments on lines 8–9 of the same figure. Likewise, the `publishPresenceEvery($)` macro defined on lines 46–48 is applied for all sensor nodes on line 2 of Figure 5. Macros enable reuse of quantified code blocks within and across WSN applications.

6. Instantiating CrimeSPOT on top of the LooCi Event-Based Middleware

We instantiated CrimeSPOT on top of the LooCi [14] event-based middleware for the Squawk VM [26] (i.e., SunSPOT motes). As LooCI advocates the use of loosely coupled components for programming WSN nodes, all of our code blocks are actually compiled to components rather than plain Java classes. The advantage of loosely coupled components is that they can be replaced at run-time. In addition, the middleware takes care of over-the-air deployment and the routing of events over a decentralized event bus. We inherit all characteristics regarding memory footprint and event dissemination from this middleware. The resulting CrimeSPOT instance is freely available online [6].

As mentioned above, network-centric CrimeSPOT code is compiled into node-specific code. This code includes a CrimeSPOT runtime and code that will populate the runtime's rule base and configuration base. Figure 9 illustrates the phases in this compilation process:

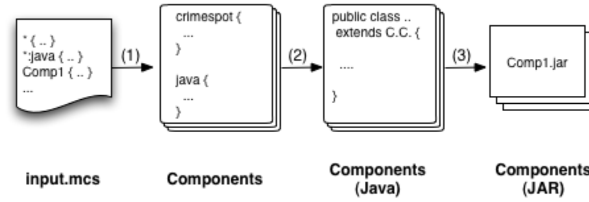

Figure 9: Compiling network-centric CrimeSPOT code to the LooCi event-driven middleware.

- First, the network-centric code is parsed to extract the quantified code blocks for every component. In case a block quantified over multiple components, its contents will be joined with the code of these components. The code from any imported libraries is included as well.
- Next, every component is processed individually. Macros and macro variables are expanded. A run-time dispatcher method is generated for the methods that can be invoked from within a component's interaction rules. This precludes the need for Java reflection, which is limited on the Squawk VM [26]. Java statements are generated that will populate the CrimeSPOT runtime that resides within each component. The generated and user-specified Java code is merged into a full-fledged LooCi component.
- Finally, the compiler produces a JAR file for every component. These JAR files can be readily deployed on WSN nodes using LooCi's over-the-air deployment facilities [14].

7. Domain-specific Extensions to the Crime Inference Engine

Section 3.2 introduced Crime [22] as the foundation of CrimeSPOT's inference engine. Before detailing how Crime can be extended with domain-specific features such as expiration and subsumption of reified events, we briefly outline its particular implementation of the Rete incremental forward chaining algorithm [9].

7.1. The Rete Algorithm as Implemented by Crime

Rete-based inference engines represent their rule base as a directed acyclic graph of computational nodes. The algorithm [9] propagates tokens, which encapsulate a set of facts, from the network's root to a terminal node. Such a path corresponds to a rule activation. Intermediate nodes may remember the tokens that pass through. These memories are at the basis of the algorithm's incremental nature. Figure 10 depicts the Rete network that corresponds to the following rule:

```
temperatureInTentNamed(Celsius=?temp,Node=?name)
  <- temperature(Celsius=?temp)@[from(MAC=?mac)],
     online(Tent=?tnt)@[from(MAC=?mac)],
     tentNamed(Tent=?tnt,Name=?name).
```

The upper and bottom part of the network are called the α-network and the β-network respectively. The former consists solely of filter nodes (depicted as triangles), while the latter consists of join nodes (depicted as rounded rectangles) and production nodes (depicted as rectangles). Tokens are depicted as the set of facts they encapsulate, demarcated by $<$ and $>$. As evidenced by the nodes' memories, some tokens have already been propagated through the network.

A token passes through a filter node if it satisfies the node's constraints. The α-network connects two such filter nodes for every condition in the body of a rule. The first filter node verifies whether the token encapsulates a single fact of which the functor agrees with the functor of the condition. The second filter node verifies whether the named attributes of the encapsulated fact agree with those of the condition. This entails verifying whether the attributes are present and whether the bindings for all occurrences of the same variable in the condition agree. A token encapsulating the fact `temperature(Fahrenheit=50)` would therefore pass through the first filter node for the `temperature` condition above, but not through the second.

Rules share the filter nodes for a common condition. As illustrated by memories T1 and T2 in Figure 10, Crime remembers the tokens that make it through a filter node. While not strictly necessary, these memories preclude the

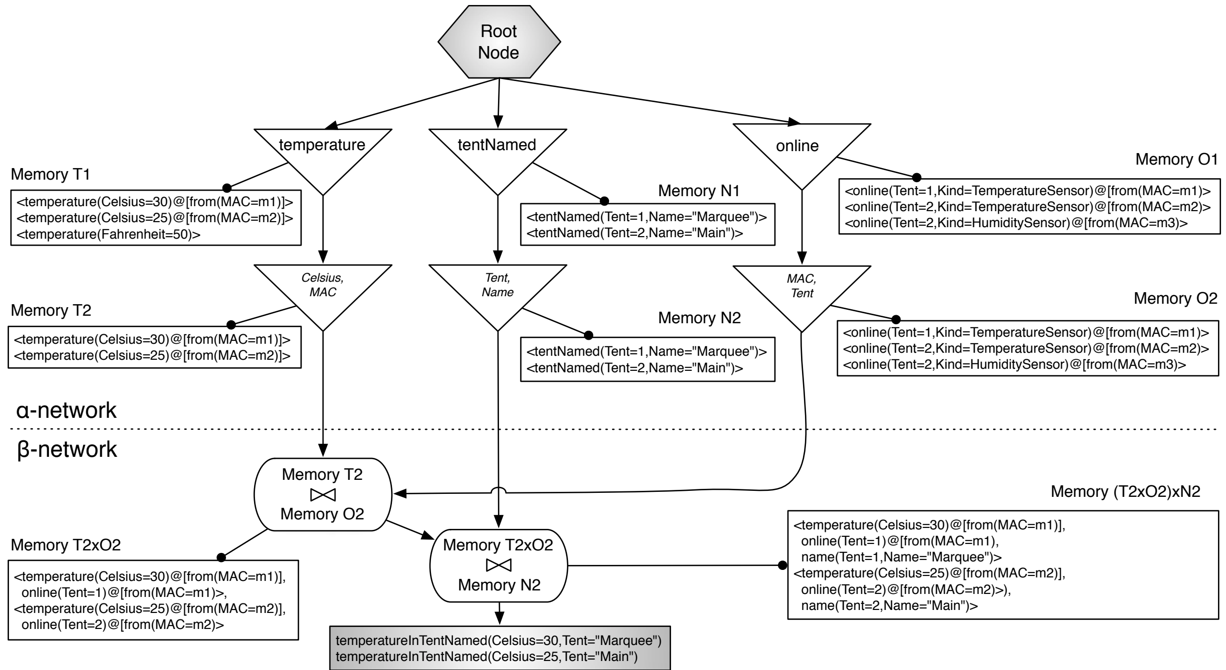

Figure 10: RETE network for a single rule that asserts `temperatureInTent` facts.

need to re-filter the entire fact base when the rule base is extended at run-time with a rule that shares existing filter nodes.

Every join node in the β-network has two parent nodes. Whenever one parent propagates a token, the join node will attempt to join this token with every token in the memory of its other parent. For a pair of tokens to be joinable, their variable bindings have to unify. For instance, when the `<temperature(Celsius=30)@[from(MAC=m1)]>` token is propagated from T2 (i.e., the memory that corresponds to matches for condition `temperature(Celsius=?temp)@[from(MAC=?mac)]`) to the leftmost join node in Figure 10, it will be joined with every token from O2 (i.e., the memory that corresponds to matches for condition `online(Tent=?tnt)@[from(MAC=?mac)]`) that has variable `?mac` bound to `m1`. This is only the case for the `<online(Tent=1))@[from(MAC=m1)]>` token from O2. Note that every pair of joinable tokens is merged into a single token, cached in the memory of the join node and further propagated through the network.

If a token reaches a production node in the β-network, a match has been found for every condition in a rule. As a result, the rule is activated for this token. For instance, whenever the production node at the bottom of Figure 10 receives a token, it will assert a `temperatureInTentNamed(Celsius=?temp,Node=?name)` fact instantiated with the token's variable bindings.

CRIME implements fact retraction by propagating a negative token through the network. The network's nodes do not remember the negative tokens that pass through. Instead, they remove any positive token from their memories that corresponds to the passing negative one. When a negative token reaches a production node, a rule has lost a match for the corresponding positive token (cf. Section 3.1). For the example rule, this entails retracting a `temperatureInTentNamed` fact from the fact base.

An agenda of operations sequentializes network accesses. The agenda is processed in a first-in first-out order. For instance, the aforementioned asserting or retracting of a fact constitutes an operation. When processed, it will encapsulate the fact in a positive or negative token and hand this token to the root node of the network. This particular operation is, among others, added to the agenda whenever a new event is received and whenever a token reaches a production node.

7.2. *Time-related Extensions to the* CRIME *Inference Engine*

We discuss the implementation of fact expiration, match expiration and match verification first. To support these time-related CRIMESPOT features, we introduce a timer that can be used to schedule various manipulations of the aforementioned agenda (i.e., addition or removal of particular operations).

Upon the assertion of a fact that carries a `factExpires` declaration in its meta-data (cf. Section 3.3), an agenda operation is scheduled with the timer to retract this fact after the specified amount of time has passed. Note that such a fact may be retracted before it expires (e.g., if the rule it originated from lost a match). In this case, we instruct the timer to unschedule the retraction activation.

To support `matchEvery` declarations among the meta-data of conditions (cf. Section 4.3), we extend the filter nodes of the RETE network with filter actions. 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 actions with this token. Such a filter action is added to the second filter node of every condition that carries a `matchEvery` declaration. Upon initialization, a timed-out operation is scheduled to be added to the agenda after the specified amount of time has passed. Processing this timed-out operation asserts a `timedOut` fact for the condition (cf. Section 4.3). This fact is retracted by the node's filter action whenever it is invoked with a positive token (i.e., whenever the condition obtains a match) within the required time limit. Regardless of whether it was invoked timely, the filter action also reschedule the timed-out operation with the timer.

We add another filter action to the second filter node of every condition that carries a `matchExpires` declaration among its meta-data (cf. Section 4.4). Whenever this filter action is invoked with a positive token, it schedules a token insertion operation to be added to the agenda after the specified amount of time. This operation propagates a negative version[2] of the positive token through the network, causing the corresponding match to expire. Whenever the aforementioned filter action is invoked with a negative token, it will unschedule the token insertion operation. This is because a condition can lose a match before it expired according to its `matchExpires` declaration (e.g., when a fact is retracted).

7.3. *Invocation-related Extensions to the* CRIME *Inference Engine*

Finally, CRIMESPOT supports invoking application logic from within the body of a rule through invocation conditions of the form "`<variable> is <invocation>`" (cf. Section 4.5). To this end, we extend the network of CRIME's inference engine with a new kind of node: invocation nodes. Invocation nodes without a parent correspond to the first condition in a rule. Other invocation nodes do have a parent. We discuss these two cases separately.

7.3.1. *Invocation Nodes without a Parent*

An invocation node without a parent behaves similarly to a filter node. It filters the results of method invocations and only allows those results to pass that unify with the expression on the left-hand side of the `is`-keyword. Unlike a filter node, however, such invocation nodes are not connected to the root node of the RETE network. This is because a parent-less invocation node is independent of the fact base. As soon as the node is initialized, it will invoke its method. If the result unifies with the left-hand side of the condition, the result is encapsulated in a token that is remembered and passed to the child of the invocation node.

For invocation conditions with a `renewEvery` or an `evalEvery` declaration among their meta-data, the method has to be invoked multiple times. In this case, an invocation operation will be scheduled with the timer (cf. Section 7.2) to be added to the agenda at the specified interval (e.g., every 10 seconds). Such an operation will trigger the invocation node. Adorned with an `evalEvery` declaration, the invocation node will just repeat the same behavior when it is triggered. Adorned with a `renewEvery` declaration, the invocation node will first remove the stored token from its memory and pass the negated version to its child node (i.e., to invalidate the previous match for the condition) before repeating the aforementioned behavior.

[2]In RETE-terms, a match for a condition corresponds to the insertion of a positive token in the conditions second filter node, while the invalidation of a match corresponds to the insertion of a negative token.

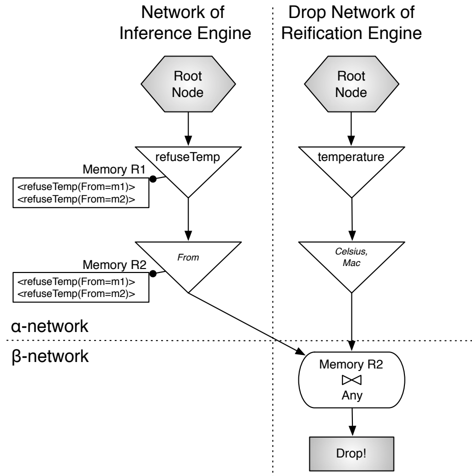

Figure 11: RETE network for a drop declaration as implemented by the reification engine of CRIMESPOT.

7.3.2. *Invocation Nodes with a Parent*

Invocation nodes with a parent are triggered as soon as their parent passes a token. If a positive token is passed, the node will use this token to invoke its method (i.e., to extract values for the variables that were given as arguments to the method) and to verify whether the result unifies with the left-hand side of the `is`-keyword. If successful, the node will merge the token from its parent with the invocation result. The resulting token is stored in the node's memory and propagated to the child node.

If a negative token is passed, the node should invalidate a previously propagated invocation result rather than re-invoke its method. To this end, the node will search its memory for the token that encapsulates the result of the invocation that was performed using the positive version of the negative token. This invocation result is removed from the node's memory and a negated version is propagated to the child node.

In case the invocation condition specified a scheduling option, the invocation node has to invoke its method multiple times for every positive token it is passed. Therefore, for every such token, an invocation operation is scheduled with the timer (cf. Section 7.2) to be added to the agenda at the specified interval. It behaves similarly to the previous invocation operations, except that it triggers the corresponding invocation node using the token that was passed to the node. When a negative token reaches the invocation node, the operation that corresponds to the positive token is unscheduled.

7.4. *Reification-related Extensions to the* CRIME *Inference Engine*

The reification engine of the CRIMESPOT runtime (cf. Section 3.3) reifies events as facts following the process depicted in Figure 4. We implement this process by adding three operations to the agenda of the RETE-network (cf. Section 7.1) after an event has been reified as a fact: one for verifying whether the new fact should be dropped, one for retracting all facts that are subsumed by the new fact, and one for asserting the new fact into the fact base. As the latter is straightforward, we discuss the implementation of the drop and subsumption operations.

7.4.1. *Drop Operations*

The drop operation will, when processed, encapsulate the new fact in a token that is passed to the root note of a special-purpose RETE-network. Figure 11 depicts the network that corresponds to the following drop declaration:

```
drop temperature(Celsius=?n)@[from(MAC=?m)] provided refuseTemp(From=?m)
```

The conditions of the drop declaration (i.e., everything after `provided`) are transformed like the conditions of a rule and added to the RETE-network of the inference engine (cf. Section 7.1). The incoming fact of the drop declaration (i.e., everything between `drop` and `provided`) is transformed as a condition as well, but added to the RETE-network of the reification engine.

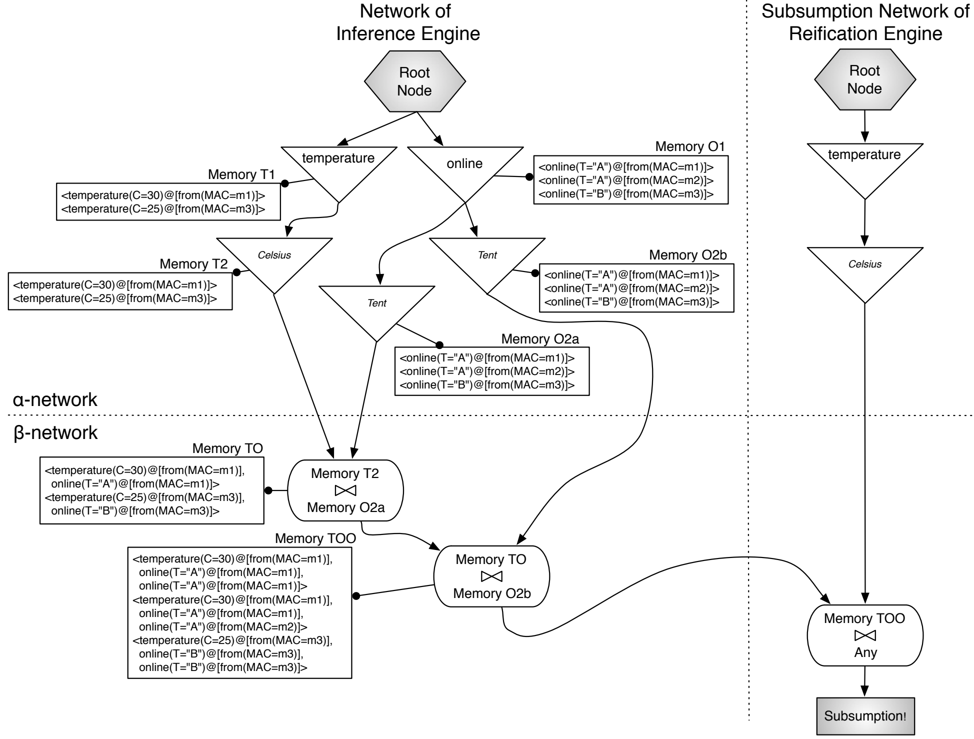

Figure 12: Rete network for a subsumption declaration as implemented by the reification engine of CrimeSPOT.

Note that the nodes in the reification network differ from those in the inference network. First of all, the filter and join nodes do not have to remember any tokens. Furthermore, the join node only has to join the newly inserted token from its reification parent with the tokens from its inference parent. As soon as one pair of tokens has been joined successfully, the node stops the joining process and propagates the merged token immediately.

Consider receiving a new `temperature(Celsius=-1)@[from(MAC=m2)]` fact. The drop operation will encapsulate this fact in a token and pass it to the root node of the drop network of the reification engine. There, the token will pass through the first two filter nodes and reach the join node where it will be joined successfully with token `<refuseTemp(From=m2)>` from memory R2. As a result, a `<refuseTemp(From=m2),temperature(Celsius=-1)@[from(MAC=m2)]>` token will be passed to the production node. The latter immediately removes two operations from the agenda: the one that would have retracted all subsumed facts and the one that would have asserted the new fact. As the new fact is to be dropped, these operations should no longer be processed. Had the token not reached the production node, these operations would have been processed one after another as soon as token propagation was complete.

7.4.2. Subsumption Operations

The drop operation will, when processed, encapsulate the fact it was created for in a token and pass this token to the root of a special-purpose Rete network. Figure 12 depicts the network that corresponds to the following subsumption declaration:

```
incoming temperature(Celsius=?new)@[from(MAC=?m)]
subsumes temperature(Celsius=?old)@[from(MAC=?otherm)]
provided online(Tent=?tnt)@[from(MAC=?otherm)],
         online(Tent=?tnt)@[from(MAC=?m)]
```

The conditions of this declaration (i.e., everything after `provided`) are first merged with the specification of the fact that is to be subsumed (i.e., everything between `subsumes` and `provided`). The result is transformed like the conditions

```
1.    * {
2.        defvar $detectionInterval: Seconds=60.
3.        defvar $alarmThreshold: 2.
4.    }
5.
6.    Controller {
7.        mySensor(MAC=MAC1).
8.        mySensor(MAC=MAC2).
9.
10.       itsController()@[to(MAC=?sensorMAC)]
11.         <- mySensor(MAC=?sensorMAC).
12.
13.       this.respond
14.         <- findall(?m,
15.                    [smoke()@[from(MAC=?m)]],
16.                    ?alarmingSensors),
17.            length(?alarmingSensors,?L),
18.            ?L > $alarmThreshold.
19.   }
20.
21.   Controller.java {
22.     private CSAction respond = new CSAction() {
23.        public void activated(CSVariableBindings bindings) { //sound alarm }
24.        public void deactivate(CSVariableBindings bindings) { //silence alarm }
25.     };
26.   }
27.
28.   SmokeDetector {
29.       smoke()@[to(MAC=?mac,factExpires($detectionInterval)]
30.         <- itsController()@[from(MAC=?mac)],
31.            true is this.smokeDetected()@[renewEvery($detectionInterval)]
32.   }
33.
34.   SmokeDetector.java {
35.      private CSValue smokeDetected() { //read out smoke sensor }
36.   }
```

```
1.    TemperatureSensor {
2.      defvar $publicationInterval: Seconds=60.
3.
4.      temperatureReading(Celsius=?temp)@[to(MAC=*,factExpires($publicationInterval)]
5.          <- itsController()@[from(MAC=?mac)],
6.             ?temp is this.getTemperature()@[renewEvery($publicationInterval)].
7.
8.      this.alertMaxTemp
9.          <- maximumTemperature().
10.   }
11.
12.   TemperatureSensor.java {
13.     private CSAction alertMaxTemp = new CSAction() {
14.        public void activated(CSVariableBindings bindings) { //blink led }
15.        public void deactivate(CSVariableBindings bindings) { //led off }
16.     };
17.   }
18.
19.   AvgComputer {
20.      this.outputAverageTemp
21.          <- findall(?t,[temperatureReading(Celsius=?t)],?temps),
22.             ?avgTemp is this.computeAverage(?temps).
23.
24.      maximumTemperature(Celsius=?maxTemp)@[to(MAC=?m)]
25.          <- findall(?t,[temperatureReading(Celsius=?t)],?temps),
26.             ?maxTemp is this.computeMaximum(?temps),
27.             temperatureReading(Celsius=?maxTemp)@[from(MAC=?m)]
28.   }
29.
30.   AvgComputer.java {
31.     private CSAction outputAverageTemp = new CSAction() {
32.        public void activated(CSVariableBindings bindings) { //show on display }
33.        public void deactivate(CSVariableBindings bindings) { //erase display }
34.     };
35.     private CSValue computeAverage(CSVariableBindings bindings) { ... }
36.     private CSValue computeMaximum(CSVariableBindings bindings) { ... }
37.   }
```

Figure 13: CrimeSPOT implementation of a WSN application for fire detection (left) and temperature monitoring (right).

in a rule and added to the Rete-network of the inference engine (cf. Section 7.1). Note that the depicted network illustrates the sharing of nodes: both `online` conditions share a common filter node.

The specification of the incoming fact (i.e., everything between `incoming` and `subsumes`) is transformed like a regular condition and added to the subsumption network of the reification engine. Most nodes in this network are identical to the ones of the aforementioned drop network. However, the join node will attempt to join the incoming token from its reification parent with all tokens from its inference parent. It will consequentially propagate *all* successfully joined tokens to its the production node. The subsumed fact will be in the first position of any token that reaches the production node and will be retracted from the fact base.

Consider the drop operation for a `temperature(Celsius=28)@[from(MAC=m2)]` fact that wasn't dropped. When the token that encapsulates this fact reaches the join node, it will be joined successfully with the token `<temperature(Celsius=30)@[from(MAC=m1)], online(Tent="A")@[from(MAC=m1)], online(Tent="A")@[from(MAC=m2)]>`. As a result, the following token will reach the production node:

```
<temperature(Celsius=30)@[from(MAC=m1)],
 online(Tent="A")@[from(MAC=m1)], online(Tent="A")@[from(MAC=m2)], temperature(Celsius=28)@[from(MAC=m2)]>
```

The production node will immediately retract the subsumed fact (i.e., `temperature(Celsius=30)@[from(MAC="m1")]` in the first position of the token) from the fact base. Note that the overhead of propagating a token through the subsumption network of the reification engine is miminal. All intermediate results from the network of the inference engine are cached and don't have to be recomputed. The same goes for propagating a token through the drop network of the reification engine.

8. Evaluation

Wireless sensor networks support a plethora of applications. Examples include scientific monitoring (e.g., monitoring wildlife habitats [19], zebras [16] and glaciers [20]), detecting emergency situations (e.g., detecting intrusions [2], forest fires [12] and river floodings [15]), as well as more active ones such as controlling heating- and air conditioning systems [7].

Using several small, but representative WSN applications, this section evaluates the expressiveness of CrimeSPOT and the overhead of its accompanying runtime —instantiated on top of the LooCi [14] event-based middleware for the Squawk VM [26] (cf. Section 6). In addition to the CrimeSPOT implementation of the motivating example depicted in Figure 5 and Figure 6, we provide a CrimeSPOT implementation for the following applications:

```
1.    RangeComponent {
2.        defvar $rangeBroadcastInterval: Seconds=3600.
3.
4.        incoming sensorRange(X=?x,Y=?y,R=?r)@[from(MAC=?m)]
5.        subsumes sensorRange(X=?ox,Y=?oy,R=?or)@[from(MAC=?m)].
6.
7.        sensorRange(X=?x,Y=?y,R=?r)@[to(MAC=*),factExpires($rangeBroadcastInterval)]
8.          <- ?x is this.getX()@[renewEvery($rangeBroadcastInterval)],
9.             ?y is this.getY(),
10.            ?r is this.getR().
11.
12.       defmacro verifyPartialCoverage($verificationMethod,$factToAssert):
13.          $factToAssert(byMAC=?m)
14.             <- sensorRange(X=?x,Y=?y,R=?r)@[from(MAC=?m)],
15.                ?m != this.MAC,
16.                true is this.$verificationMethod(?x,?y,?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 CSValue 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 CSBooleanValue(true);
40.           else
41.               return CSBooleanValue(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.   }
```

```
1.    Logger, Logger.java, RiverMonitor.java {
2.        //not shown, merely have to invoke application logic
3.    }
4.
5.    RiverMonitor {
6.        defvar $logger: MACLogger.
7.        defvar $verifyRiverLevelInterval: Seconds=600.
8.        defvar $floodThreshold: 20.
9.        defvar $verifyTheftInterval: Seconds=60.
10.       defvar $theftThreshold: 1.
11.
12.       riverLevel(L=?l)@[to(MAC=$logger),factExpires($verifyRiverLevelInterval)]
13.         <- ?l is this.getRiverLevel()@[renewEvery($verifyRiverLevelInterval)].
14.
15.       possibleTheft()@[to(MAC=$logger)]
16.         <- ?a is this.getAcceleration()@[renewEvery($verifyTheftInterval)],
17.            ?a > $theftThreshold.
18.
19.       this.controlSluice
20.         <- l is this.getRiverLevel()@[renewEvery($verifyRiverLevelInterval)],
21.            ?l > $floodThreshold.
22.   }
```

Figure 14: CrimeSPOT implementation of a WSN application for range coverage (left) and river monitoring (right).

Fire detection using multiple *SmokeDetector* and multiple *Controller* components. The former report their sensor readings to the controllers in their neighborhood (i.e., a sensor can report to multiple controllers) at a predefined rate. As soon as a particular *SmokeDetector* has reported several (i.e., above a certain threshold) consecutive smoke incidents to the same *Controller*, the controller sounds an alarm. The alarm is silenced when the threshold is no longer met. The left-hand side of Figure 13 depicts the CrimeSPOT implementation of this application.

Temperature monitoring using multiple *TemperatureSensor* components and a single *AverageComputer* component. The latter continuously displays the average of the latest temperature readings it received from all of the former. In addition, the former blink a led if they are currently measuring the highest temperature of all. The right-hand side of Figure 13 depicts the CrimeSPOT implementation of this application.

Range coverage using several *Range* components that determine whether their node's sensing range is covered by other nodes in the neighborhood. The implementation supports extending existing CrimeSPOT applications with functionality for range coverage. The left-hand side of Figure 14 depicts the CrimeSPOT implementation of this application.

River and theft monitoring using several *RiverMonitor* components and a *Logger* component. The former monitor river levels, control sluices and warn about possible thefts. The latter logs the reported river levels in a centralized manner. The right-hand side of Figure 14 depicts the CrimeSPOT implementation of this application.

Figure 15: CrimeSPOT micro-benchmarks for performance and memory overhead.

8.1. Expressiveness of the Language

Each application required on average 2.11 components (min: 2, max: 4). The average component has about 0.22 declarations for the reification engine (min: 0, max: 4), 4.14 interaction rules (min: 3, max: 14) and 3.72 component methods (min: 4, max:10). This is testament to the conciseness of CrimeSPOT applications. More code was required for the motivating example of this paper due to the complexity of its functional requirements.

A substantial amount of code would be required to implement these WSN applications on top of event-based middleware —let alone plain Java. This is already the case for much simpler applications. In order to assess this burden on developers, we implemented the equivalent of the following CrimeSPOT toy application in Java using the SunSPOT SDK:

```
* { pong(value=?x)@[to(MAC=*)] <- ping(value=?x).
    ping(value=?value)@[to(MAC=*)] <- ?value is this.getLightReading()@[evalEvery(Seconds=1)]. }
```

The CrimeSPOT application consist of two straightforward interaction rules that are deployed on every WSN mote. The first rule publishes a pong fact to all motes whenever a ping fact is received. The second rule continuously reads out the mote's light sensor and publishes a ping fact with this reading.

An equivalent Java implementation using the SunSPOT SDK is listed in Appendix A. This code is not only verbose (about 101 lines), but it is also complex. Several low-level concerns can be discerned. Interestingly, only 3 lines of code are dedicated to application logic. A significant amount of bookkeeping is required to deal with error handling (21 Loc), packaging the data before it can be sent over the network (12 Loc), multithreading (10 Loc), and low-level communication primitives (9 Loc).

Note that the Java implementation does not yet store the events it receives, only dispatches over a single event type in its event handler, and does not match the payload of its outgoing pong event to the payload of the incoming ping event. This would require even more code. More importantly, ad-hoc implementations of such event *dispatching*, *storage* and *matching* would be duplicated across the event handlers of each WSN node. The same goes for the event *expiration*, *subsumption* and *compensation* required by our motivating example. Finally, the resulting event handlers would be difficult to compose.

8.2. Overhead of the Runtime

The price to pay for the aforementioned domain-specific language support and their software engineering benefits is reasonable. We conducted the micro-benchmarks discussed below on standard SunSPOT motes (180MHz ARM9 CPU, 512KB RAM, 4MB Flash, SQUAWK VM version RED-100104).

Memory Overhead. Our runtime requires about 460kB of ROM (i.e., 9.7% of the available flash memory). This is to be expected as we made no conscious effort to reduce this footprint at all. There is therefore ample room for improvement.

At run-time, every asserted fact consumes about 3kB of RAM. The amount of RAM that is consumed by a rule depends on the complexity of the corresponding Rete network. This is illustrated by the right-most graph of Figure 15. It depicts the maximum amount of distinct rules that can be added to the rule base of a SunSPOT mote before it runs

out of memory. The horizontal axis indicates the number of conditions each rule contains, while the vertical axis indicates how many rules of this type can be deployed. For example, it is possible to deploy 71 rules of which the body consists of one condition. All rules are of the following form:

```
fact(A=?x, B=?y) <-  c1(A=?x, B=?y), c2(A=?x, B=?y), ...
```

Such rules represent the *worst-case situation* in terms of memory consumption as no filter nodes can be reused in their corresponding RETE networks. Note that none of our representative WSN applications required more than 6 conditions in a rule. However, there is ample room for improvement. For instance, our implementation of the RETE network is completely object-oriented.

Performance Overhead. It takes about 80ms on average for a received fact to be added to the local fact base. The time it takes to react to such a fact depends on the complexity of the RETE network it has to be processed by. However, we performed several micro-benchmarks to give an indication of the performance of our run-time. In each micro-benchmark, we sequentially added a certain amount of facts one by one to the fact base and re-computed all of the matches for a single rule in between each addition.

In the first benchmark, we used a rule that consists of 2 conditions of the aforementioned form. Rules of this form stress the runtime as their conditions are highly correlated (i.e., their corresponding attributes have to be matched). The left-most graph of Figure 15 depicts the results. The vertical axis indicates the time required to process the amount of facts indicated on the horizontal axis. The dark line plots the time required by a naive Java implementation using the SUNSPOT SDK. This implementation stores all facts in a collection and re-computes all pairs of matching facts after each addition (i.e., the equivalent rule consists of two conditions). Note that the RETE network outperforms the naive Java implementation when more than 50 facts have to matched against this two-condition rule.

The graph in the middle of Figure 15 depicts the performance overhead for a similar rule of which the body consists of twenty conditions. Although all of their corresponding attributes have to match, as before, the processing time does not increase significantly. This is due to the RETE network's caching of partial matches.

To conclude, the processing capabilities of the SUNSPOT motes are more than adequate to support our runtime. In this regard, these motes are situated at the high-end of the WSN market. However, we firmly believe that the software engineering benefits brought by CRIMESPOT will outweigh the cost of such nodes as the complexity of WSN applications increases.

9. Limitations and Future Work

Some event-driven middleware supports hierarchical relations between events. This enables an event handler to react to an event type as well as its subtypes. The CRIMESPOT runtime does not consider such relations in its matching of facts with the conditions of a rule (cf. Section 4). For a fact and a condition to match, their functors have to be the same. This might lead developers to duplicate an interaction rule such that it reacts to event subtypes. In theory, quantified code blocks of the following form could encode the hierarchical relations between middleware events:

```
* { superEvent(SuperAttribute=?a) <- subEvent(SuperAttribute=?a) }
```

Note that CRIMESPOT allows `subEvent` facts to exhibit more attributes than enumerated in the condition. However, there is no means to transfer the meta-data associated with the `subEvent` fact in the body of the rule (e.g., fact expiration or origin) to the `superEvent` fact in the head of the rule. We intend to address this limitation in future work. Alternatively, we could enable developers to model the hierarchical semantics of events explicitly using ontologies expressed in a standard semantic language such as OWL. There is already a significant body of work on incorporating OWL in RETE-based inference engines (e.g., [8, 21]).

Another limitation of the CRIMESPOT prototype concerns its support for expressing temporal relations between events. Currently, these have to be specified in a rather operational manner through arithmetic constraints on timing-related meta-data of facts. We intend to investigate more declarative specification means in future work. Some of the authors have already incorporated the past-oriented subset of metric temporal logic [13] in a RETE-based inference engine. Similarly, Walzer et al. [29] have incorporated Allen's operators for describing the temporal relations between time intervals. Both approaches employ a "garbage collection" strategy that removes tokens from the RETE network

when they can no longer contribute to matches for a temporal relation. It would be interesting to adapt these strategies to CrimeSPOT's provisions for fact subsumption and expiration.

In future work, we will investigate how developers can exert more control over the causality tracking that allows reacting to rules that lose a match (e.g., when a fact expires). While desirable in most WSN situations, this tracking does cause an overhead for facts that are extremely short-lived. Along the same lines, we intend to investigate how more control can be offered over the order in which rules with a common body are activated. Currently, the activation precedence of rules is determined by the order in which they are specified. Finally, CrimeSPOT does not offer facilities for publishing a fact to the physical *n*-hop neighborhood of a node. Facts that have a `to(MAC=*)` declaration among their meta-data are assumed to be network-wide. It would be interesting to investigate language support for changing a fact as it traverses physical hops.

10. Related Work

We refer the reader to an excellent 2011 survey [24] for a complete overview of the state of the art in programming wireless sensor networks. In this section, we limit our discussion to those approaches that are most closely related. In general, each approach can be categorized as either node-centric or network-centric [27]. The overall goal of CrimeSPOT is to bring *node-centric* programming of *active* WSNs closer to *network-centric* programming of *passive* WSNs.

We discuss the closely related node-centric approaches first. Logical Neighborhoods [23] advocates sending messages to logically specified groups of nodes in the network. The way in which we addressed groups of nodes is less declarative and hence open to similar improvements. TeenyLIME [5] allows neighboring nodes to interact by storing tuples in a shared tuple space. However, both approaches require an event handler to react to incoming events. The rule-based language FACTS [28] comes closest to the node-centric features of CrimeSPOT. It allows nodes to interact by exchanging facts. These facts can be reacted to through declarative rules. However, logic variables cannot be used within these rules. As a result, a node cannot react to several related facts. In addition, facts cannot be declared to expire. As the causality between bodies and heads is not tracked, rule deactivation cannot be reacted to either.

The network-centric features of CrimeSPOT are comparable to those introduced by ATaG [3]. ATaG advocates specifying a WSN application in terms of tasks that have to be instantiated on particular nodes. Unlike CrimeSPOT, ATaG employs a graphical notation and is more expressive concerning the instantiation of tasks on nodes and the interactions between tasks. However, ATaG provides no support for programming the tasks themselves. Reactions to incoming data still have to be implemented through an event handler. Moreover, there is no control over the subsumption and expiration of this data.

Distributed rule-based systems have also been applied outside of the WSN domain. Most notably, to render ambient intelligence applications aware of their context. In these settings, events typically carry context information. Chisel [17] is a rule-based framework for Java that enables applications to adapt their behavior in response to changes to their surroundings. Adaptations can be performed at runtime using reflection, and their functional requirements can be controlled through meta-types. Contrary to CrimeSPOT, Chisel's policy model does not allow for sharing of context information with neighbors. Cocoa [1] defines context as the union of all contextual views of the entities within a well defined distance. The ability to share contextual information between entities and react upon these in a declarative scripting language make it a powerful framework. Contrary to CrimeSPOT it does not give a meaningful semantics to the retraction of information. Gaia [25] is a framework for building context-aware applications. The focus of Gaia lies in the derivation of higher-order contexts through declarative rules and machine learning techniques. Gaia enables context clients to reason about past contexts in addition to the current one. As discussed before, our CrimeSPOT prototype does not yet support expressing temporal relations between events.

DJess [4] (distributed Jess) is an extension of Jess [10] allowing it to be used as a lightweight middleware for sharing contextual knowledge. The current locking mechanism of DJess is not resilient to failing nodes: if a fact is locked by a certain process and that process dies, the fact remains locked forever. In CrimeSPOT, facts are shared by copy and thus no locking is required. Building upon approaches such as Gaia and DJess, García-Herranz et al. [11] propose high-level language abstractions that enable end-users to develop ambient intelligence applications from a network-centric perspective. It would be interesting to investigate end-user programming of active WSN applications on top of CrimeSPOT as well.

11. Conclusion

In this paper, we presented CrimeSPOT as a domain-specific language that minimizes the accidental complexity inherent to programming WSN applications using event-based middleware. We carefully motivated the need for such a language through a motivating example. This example is representative for applications in which nodes are not only tasked with sensing, but also with reacting to sensor readings. Having introduced the runtime that supports CrimeSPOT, we provided a comprehensive overview its node-centric and network-centric features. In addition, we detailed the implementation of its accompanying runtime. The latter comprises several extensions to the Rete forward chaining algorithm. Through five example applications and some illustrative micro-benchmarks, we evaluated the expressiveness of this language and the overhead of its supporting runtime as instantiated on the LooCi [14] event-based middleware. The resulting prototype implementation is freely available.

Acknowledgments

The authors thank everyone who contributed to the Crime inference engine upon which CrimeSPOT builds: Eline Philips and Stijn Mostinckx. Coen De Roover is funded by the *Stadium* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen).

Appendix A. SunSPOT SDK Implementation of the CRIMESPOT Toy Application

```
package edu.vub.soft.crime;
import javax.microedition.io.Connector; //SUNPSOT Specific
import javax.microedition.io.Datagram;  //SUNPSOT Specific
import javax.microedition.midlet.MIDlet; //SUNPSOT Specific
import javax.microedition.midlet.MIDletStateChangeException; //SUNPSOT Specific
import com.sun.spot.io.j2me.radiogram.RadiogramConnection;   //SUNPSOT Specific

class Receive extends Thread { //multithreading
        private int port_; //communication
        private NaiveSendReceive s_;

        Receive(int port, NaiveSendReceive s) {
                port_ = port;
                s_ = s;
        }

        public void run()  { //multithreading
        RadiogramConnection rCon = null; //communication
        Datagram dg = null;

        try { //error handling
            rCon = (RadiogramConnection) Connector.open("radiogram://:" + port_); //communication
            dg = rCon.newDatagram(rCon.getMaximumLength()); //data packaging
        } catch (Exception e) { //error handling
             System.err.println("setUp caught " + e.getMessage());//error handling
        }//error handling

        // Main data collection loop
        while (true) {
            try { //error handling
                rCon.receive(dg); //communication
                String addr = dg.getAddress();  //data packaging
                int val = dg.readInt();         //data packaging
                System.out.println( "from: " + addr + "   value = " + val);
                if( val == NaiveSendReceive.PING ) { //application logic
                        s_.pong(); //application logic
                }
            } catch (Exception e) { //error handling
                System.err.println("Caught " + e +  " while reading sensor samples.");//error handling
            }//error handling
        }
    }
}

public class NaiveSendReceive extends MIDlet { //SUNPSOT Specific
    private static final int HOST_PORT = 67; //communication
    static final int PING = 42; //data packaging
    static final int PONG = 84; //data packaging
    private RadiogramConnection rCon = null; //communication
    private Datagram dg = null;//data packaging

    synchronized void pong() {    //multithreading
            try {//error handling
                    dg.reset(); //data packaging
            dg.writeInt(PONG); //data packaging
            rCon.send(dg); //communication
         } catch (Exception e) { //error handling
            System.err.println("Caught " + e + " while collecting/sending sensor sample.");//error handling
        }//error handling
    }

    synchronized void ping() {   //multithreading
            try { //error handling
                    dg.reset(); //data packaging
            dg.writeInt(PING); //data packaging
            rCon.send(dg); //communication
        } catch (Exception e) {//error handling
            System.err.println("Caught " + e + " while collecting/sending sensor sample.");//error handling
        }//error handling
    }

    protected void startApp() throws MIDletStateChangeException {  ////Multi threading, SUNPSOT Specific
        // Listen for downloads/commands over USB connection
        new com.sun.spot.util.BootloaderListener().start();    //SUNPSOT Specific

        //opening a broadcast channel
        try {//error handling
            rCon = (RadiogramConnection) Connector.open("radiogram://broadcast:" + HOST_PORT); //communication
            // only sending 12 bytes of data
            dg = rCon.newDatagram(50);   //data packaging
        } catch (Exception e) {//error handling
            System.err.println("Caught " + e + " in connection initialization.");//error handling
            System.exit(1);//error handling
        }//error handling

        Receive r = new Receive(HOST_PORT,this); //multi threading
            r.start();     //multi threading

            while(true) {  //multi threading
                    ping();  //application logic
            }
    }

    protected void pauseApp() { //multi threading, SUNPSOT Specific
        // This will never be called by the Squawk VM
    }

    protected void destroyApp(boolean arg0) throws MIDletStateChangeException { //multi threading, SUNPSOT Specific
        // Only called if startApp throws any exception other than MIDletStateChangeException
    }
}
```

References

[1] And, O. H., 1999. Stigmergy, self-organisation, and sorting in collective robotics. Artificial Life, 173–202.

[2] Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y., Herman, T., Kulkarni, S., Arumugam, U., Nesterenko, M., Vora, A., Miyashita, M., 2004. A line in the sand: a wireless sensor network for target detection, classification, and tracking. Computer Networks 46 (5), 605–634.

[3] Bakshi, A., Prasanna, V. K., Reich, J., Larner, D., 2005. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In: Proceedings of the 2005 Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services (EESR05). pp. 19–24.

[4] Cabitza, F., Sarini, M., Seno, B. D., 2005. Djess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In: International Conference on Pervasive Services (ICPS05). pp. 229–238.

[5] Costa, P., Mottola, L., Murphy, A. L., Picco, G. P., 2007. Programming wireless sensor networks with the TeenyLime middleware. In: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware (MIDDLEWARE07). pp. 429–449.

[6] CrimeSPOT Team, 2011. CrimeSPOT website. http://soft.vub.ac.be/amop/crime/sunspot.

[7] Deshpande, A., Guestrin, C., Madden, S. R., 2005. Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28 (1).

[8] Dunkel, J., Fernández, A., Ortiz, R., Ossowski, S., 2009. Injecting semantics into event-driven architectures. In: Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS09). pp. 70–75.

[9] Forgy, C., 1982. Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence 19 (1), 17–37.

[10] Friedman-Hill, E., 2002. Jess in Action : Java Rule-Based Systems (In Action series). Manning Publications.

[11] García-Herranz, M., Haya, P. A., Alamán, X., 2010. Towards a ubiquitous end-user programming system for smart spaces. Journal of Universal Computer Science 16 (12), 1633–1649.

[12] Hartung, C., Han, R., Seielstad, C., Holbrook, S., 2006. Firewxnet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In: Proceedings of the 4th international conference on Mobile Systems, Applications and Services (MobiSys06). pp. 28–41.

[13] Herzeel, C., Gybels, K., Costanza, P., De Roover, C., D'Hondt, T., April 2009. Forward chaining in HALO: An implementation strategy for history-based logic pointcuts. Elsevier International Journal on Computer Languages, Systems & Structures 35 (1), 31–47.

[14] Hughes, D., Thoelen, K., Horré, W., Matthys, N., Cid, J. D., Michiels, S., Huygens, C., Joosen, W., 2009. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In: Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM09). pp. 195–203.

[15] Hughes, D., Thoelen, K., Horré, W., Matthys, N., del Cid, J., Michiels, S., Huygens, C., Joosen, W., Ueyama, J., 2010. Building wireless sensor network applications with looci. International Journal of Mobile Computing and Multimedia Communications 2 (4), 38–64.

[16] Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L. S., Rubenstein, D., 2002. 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 (ASPLOS02). pp. 96–107.

[17] Keeney, J., 2003. Chisel: A policy-driven, context-aware, dynamic adaptation framework. Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY03).

[18] Maier, I., Rompf, T., Odersky, M., 2010. Deprecating the observer pattern. Tech. rep., Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

[19] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., Anderson, J., 2002. Wireless sensor networks for habitat monitoring. In: Proceedings of the 1st ACM international workshop on Wireless Sensor Networks and Applications (WSNA02). pp. 88–97.

[20] Martinez, K., Hart, J. K., Ong, R., 2004. Environmental sensor networks. IEEE Computer 37 (8), 50–56.

[21] Meditskos, G., Bassiliades, N., 2008. A rule-based object-oriented owl reasoner. IEEE Transactions on Knowledge and Data Engineering 20 (3), 397–410.

[22] Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., Meuter, W. D., 2007. Fact spaces: Coordination in the face of disconnection. In: Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION07). pp. 268–285.

[23] Mottola, L., Picco, G., 2006. Logical neighborhoods: A programming abstraction for wireless sensor networks. In: Proceedings of the 2nd ACM/IEEE International Conference on Distributed Computing on Sensor Systems (DCOSS06).

[24] Mottola, L., Picco, G. P., 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Computing Surveys 43 (3), 19:1–19:51.

[25] Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., Nahrstedt, K., 2002. A middleware infrastructure for active spaces. IEEE Pervasive Computing 1 (4), 74–83.

[26] Simon, D., Cifuentes, C., Cleal, D., Daniels, J., White, D., 2006. Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In: Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE06). ACM, pp. 78–88.

[27] Sugihara, R., Gupta, R. K., 2008. Programming models for sensor networks: A survey. ACM Transactions on Sensor Networks 4 (2), 1–29.

[28] Terfloth, K., Wittenburg, G., Schiller, J., 2006. FACTS - a rule-based middleware architecture for wireless sensor networks. In: Proceedings of the 1st International Conference on Communication System Software and Middleware (COMSWARE06). pp. 1–8.

[29] Walzer, K., Breddin, T., Groch, M., 2008. Relative temporal constraints in the rete algorithm for complex event detection. In: Proceedings of the second international conference on Distributed Event-based Systems (DEBS08). pp. 147–155.