CrimeSPOT: Language Support for Programming Interactions among Wireless Sensor Network Nodes

Coen De Roover, Christophe Scholliers, Wouter Amerijckx,
Theo D'Hondt and Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel, Belgium,
(cderoove—cfscholl—wamerijc—tjdhondt—wdmeuter)@vub.ac.be

***Abstract*—An emerging breed of wireless sensor network applications tasks nodes not only with sensing, but also with reacting to sensor readings. Event-based middleware lends itself to implementing such 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.**

***Keywords*-wireless sensor networks; ubiquitous computing; programming languages**

I. 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 [1]. 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.

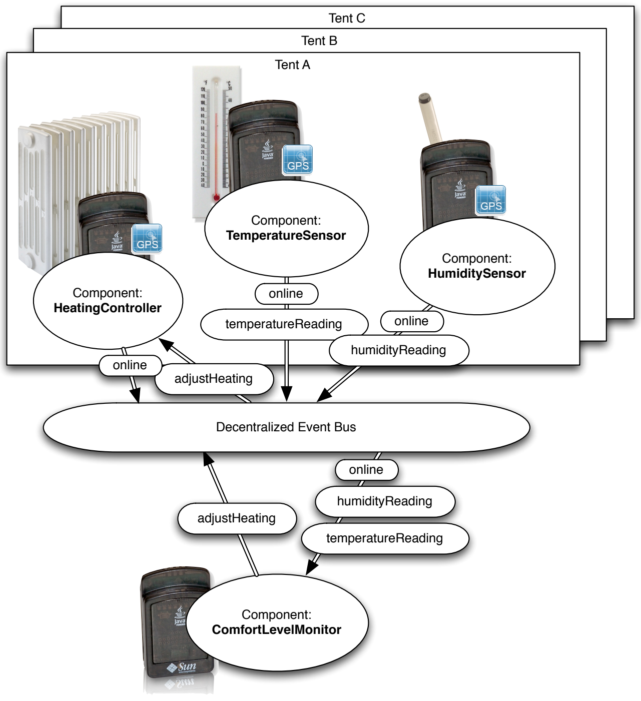

Fig. 1. Motivating example: a WSN application for festival tents.

II. 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. They 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

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.

Fig. 2. Functional requirements for the motivating example.

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.

A. 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 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.

B. 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.

III. 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 V and Section IV discuss the CRIMESPOT features it relies on from the network-centric and node-centric perspective respectively. We will discuss its accompanying runtime first.

A. *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 III-C 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,

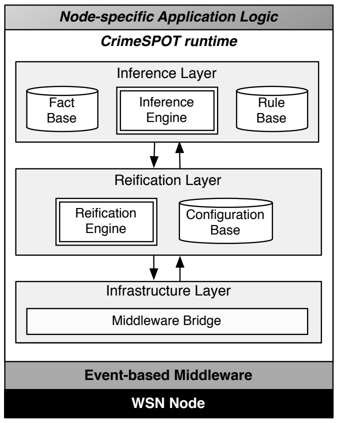

Fig. 3. Architectural overview of the CRIMESPOT runtime.

the *inference engine* re-evaluates the fact base against the rule base whenever the former changes —at least, conceptually. Section III-B 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 III-B discusses how the inference engine tracks the causality between rule bodies and heads.

B. *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 [2] 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 [3]. 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 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.

C. *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. The `factExpires(Seconds=600)` meta-data indicates that these facts are to expire after 10 minutes.

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.

[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 6).

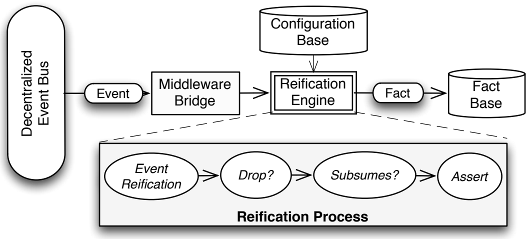

Fig. 4. The reification engine of the CRIMESPOT runtime.

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.

IV. 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. 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.

A. *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

```
TemperatureSensor, HumiditySensor, HeatingController {
    publishPresenceEvery($onlineInterval).
}

TemperatureSensor {
    temperatureMapping($readingInterval).
    temperatureReading(Celsius=?temp)@[to(MAC=*),
                                       factExpires($readingInterval)]
      <- ?temp is this.getTemperature()@[renewEvery($readingInterval)].
}

TemperatureSensor.java {
    private CSValue getTemperature() { return ... }
}

HumiditySensor {
    humidityMapping($readingInterval).
    humidityReading(Percent=?p)@[to(MAC=*),
                                 factExpires($readingInterval)]
     <- ?p is this.getHumidity()@[renewEvery($readingInterval)]
}

HumiditySensor.java {
    private CSValue getHumidity() { return ... }
}

HeatingController {
    incoming adjustHeating(Level=?new) subsumes adjustHeating(Level=?old).

    this.adjustHeater
      <- adjustHeating(Level=?h).
}

HeatingController.java {
  private CSAction adjustHeater = new CSAction() {
     public void activated(CSVariableBindings bindings) { //adjust heating }
     public void deactivate(CSVariableBindings bindings) { //reset heating }
  };
}
```

Fig. 5. CRIMESPOT code for the *TemperatureSensor*, *HumiditySensor* and *HeatingController* nodes in the motivating example.

```
1.    ComfortLevelMonitor {
2.        temperatureMapping($readingInterval).
3.        humidityMapping($readingInterval).
4.
5.        incoming online(Tent=?tnt,Component=?c)@[from(MAC=?m)]
6.        subsumes online(Tent=?otnt,Component=?c)@[from(MAC=?m)].
7.
8.        subsumesOlderFromSameTent(humidityReading,Percent).
9.        subsumesOlderFromSameTent(temperatureReading,Celsius).
10.
11.       this.logComfortLevel
12.         <- humidityReading(Percent=?h)@[from(MAC=?hm)],
13.            temperatureReading(Celsius=?t)@[from(MAC=?tm)],
14.            online(Tent=?tnt)@[from(MAC=?hm)],
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(Name=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():
43.           mapping humidityReading(Percent=?h)@[factExpires($readingInterval)]
44.              <=> Event_102(Integer=?h).
45.
46.       defmacro publishPresenceEvery($time):
47.          online(Tent=?tnt,Name=$NAME)@[to(MAC=*),factExpires($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.   }
```

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

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`:

```
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.

B. 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

18–23 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.

C. Error Handling

Some form of error handling might be in order when a match can't 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:

```
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.

D. Invoking Application Logic

Finally, CRIMESPOT supports invoking application logic from within the body or 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 [4]. 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.

V. 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. Used to specify which rules and application logic are to govern the behavior of which nodes, these 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. 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.

A. 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 [4]. 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.

B. 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.

VI. PRELIMINARY EVALUATION

We instantiated CRIMESPOT on top of the LOOCI [5] event-based middleware for the Squawk VM [4] (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 therefore inherit all characteristics regarding memory footprint and event dissemination from this middleware. The resulting CRIMESPOT instance is freely available online [6].

A. Expressiveness

We implemented several small, but representative WSN applications to evaluate the expressiveness of CRIMESPOT. Examples include active temperature and river monitoring, fire detection and dynamically measuring whether the sensor range of a node is covered by others as well. We refer to the master's thesis of Amerijckx [7] for their implementations. 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 applications in plain Java on top of event-based middleware. As argued in Section II, this would lead to adhoc and error-prone implementations of event *dispatching*, *storage* and *matching* that are duplicated across the event handlers of each node. The latter have been shown to violate several software engineering principles such as composability, scalability and separation of concerns [1]. On top of that, a significant amount of bookkeeping would be required to implement event *expiration*, *subsumption* and *compensation* as required by our motivating example.

B. Overhead

The price to pay for all this domain-specific support is reasonable. Our runtime requires about 460kB of ROM (i.e., 9.7% of the flash memory available on the SUNSPOT mote). This is to be expected as we made no conscious effort to reduce this footprint at all. There is therefore ample room for improvement. The amount of RAM that is consumed by a rule at run-time depends on the complexity of the corresponding Rete network. A rule that represents the worst-case situation for 6 conditions consumes about 30kB of RAM. In addition, every asserted fact consumes about 3kB of RAM. We attribute these numbers to the completely object-oriented implementation of the RETE network.

To give an indication about the performance of our runtime, it takes about 80ms on average for a received fact to be added to the local fact base. It then takes another 140ms before the aforementioned worst-case rule is activated in reaction to this fact. Again, the performance of our runtime is difficult to measure as it depends on the complexity of its RETE network.

The processing capabilities of the SUNSPOT motes is therefore more than adequate. In this regard, they 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.

VII. Related Work

To the best of our knowledge, CRIMESPOT is unique among the approaches that have been proposed for programming WSNs [8]. It attempts to bring node-centric programming of active WSNs closer to network-centric programming of passive WSNs.

We will discuss the most closely related node-centric approaches first. LOGICAL NEIGHBORHOODS [9] advocates sending message to logically specified groups of nodes in the network. The way in which we addressed all nodes in a festival tent is less descriptive and hence open to similar improvements. TEENYLIME [10] 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 [11] 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 [12]. 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.

VIII. Conclusion and Future Work

In this paper, we introduced 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 detailed its node-centric and network-centric features. In a preliminary evaluation, we discussed the expressiveness of this language and the overhead of its supporting runtime as instantiated on state-of-the-art middleware. The resulting prototype implementation is freely available.

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.

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). Christophe Scholliers is funded by a doctoral scholarship from the same agency. This research is partially supported by the IAP Programme of the Belgian State.

References

[1] I. Maier, T. Rompf, and M. Odersky, “Deprecating the observer pattern,” Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Tech. Rep., 2010.

[2] C. Forgy, “Rete: A fast algorithm for the many patterns/many objects match problem.” *Artificial Intelligence*, vol. 19, no. 1, pp. 17–37, 1982.

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

[4] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, “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, 2006, pp. 78–88.

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

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

[7] W. Amerijckx, “Language support for programming interactions among wireless sensor network nodes,” Master’s thesis, Vrije Universiteit Brussel, 2011.

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

[9] L. Mottola and G. Picco, “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)*, 2006.

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

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

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