Language and Middleware Support for Dynamism in Wireless Sensor and Actuator Network Applications

Nelson Matthys, Sam Michiels, and Wouter Joosen
IBBT-DistriNet
Katholieke Universiteit Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Christophe Scholliers, Coen De Roover, Wouter Amerijckx, and Theo D'Hondt
Software Languages Lab
Vrije Universiteit Brussel, Belgium
{firstname.lastname}@vub.ac.be

ABSTRACT

Advances in wireless sensing and actuation technology allow for reasonable amounts of application logic to be embedded inside wireless sensor networks. Such applications are more autonomous but are significantly more complex to program and manage. First, in-network adaptations need to be supported due to the long-lived nature of the network. Second, the intrinsic dynamism of the network challenges how applications interact with each other. Third, as applications become more complex, coordinating their interactions becomes more difficult. This paper explores how the integration of an event-based component middleware, a policy-based management system, and a domain-specific coordination language gives rise to comprehensive support for developing and managing this new breed of applications.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: [Distributed Systems, Distributed Applications]

General Terms

Design, Management, Languages

Keywords

components, policy, middleware, domain-specific languages

1. INTRODUCTION

Today's wireless sensor networks (WSN) are rapidly evolving into highly interconnected, multi-purpose infrastructure, which is equipped with a rich set of sensor and actuation functionality. From an application perspective, these advances in WSN technology introduce many opportunities for more realistic, albeit more complex applications to be developed. Compared to early WSN applications, which merely focussed on plain data collection data, this new type of applications is traditionally referred to as wireless sensor and actuator networks applications (WSANs).

WSANs can for instance be deployed in real-world scenarios such as logistics or disaster management. These scenario's require coordination between sensing, intelligent in-network processing, and actuation functionality. However, there are three main concerns that typically challenge the implementation of these scenario's: (i) functional and non-functional requirements of WSAN applications are subject to changes over time, (ii) interactions between WSAN nodes are highly volatile due to node mobility or communication failures, and (iii) as applications get more complex, coordinating these inter-node interactions becomes more difficult.

To address concern (i), our previous work proposed a run-time reconfigurable middleware [6, 9], offering components as policies as first class abstractions for application development and management. This middleware introduced a loosely-coupled, event-based style of component interaction, aligning well with the volatile nature of the network (concern (ii)) and simplifying component reconfigurations (concern (i)). However, experience has shown that programming applications in terms of events rapidly tends to become very complex, especially when multiple unordered events need to be composed together to compute the application behaviour to invoke. Hence, to overcome concern (iii), we previously proposed a domain-specific language to manage the complexity of inter-node interactions [3].

Throughout this paper, we discuss our recent efforts and experiences to combine these three individual elements into an integrated WSAN middleware: LooCI, an event-based component model for developing adaptive WSAN applications; PMA, an adaptive policy system for enforcing non-functional concerns; and CrimeSPOT, a domain-specific language to implement complex inter-node interactions.

2. EXAMPLE SCENARIO

Consider the scenario illustrated in Figure 1 in which a WSAN is used to regulate the temperature in a container. All node interactions happen asynchronously, as nodes publish events and subscribe to events using a shared decentralized event bus abstraction (cf. concern (ii)). Outgoing arrows depict events published by a node, while incoming arrows depict events a node is subscribed to. Battery-powered TemperatureSensors and HumiditySensors register environmental conditions inside the container. These readings are then used by a TransportMonitor system, employed on every storage location, which controls the individual air conditioning units of the containers by publishing `AdjustHeating` events. One HeatingController node per container serves as a wireless actuator for this air conditioning unit and

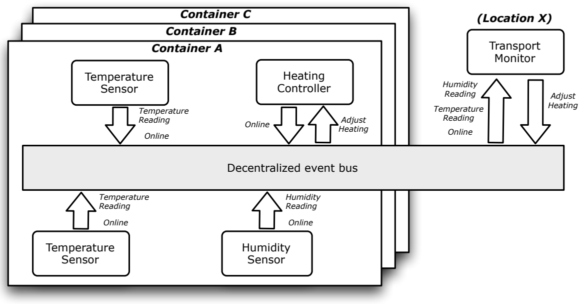

Figure 1: Motivating logistics scenario.

subscribes to `AdjustHeating` events. Upon reception of an `AdjustHeating` event, the actuator updates the settings of its air conditioning unit. Finally, every node publishes the container identifier in which it resides through an `Online` event.

2.1 Dynamic Reconfiguration

Functional and non-functional requirements of WSAN applications are subject to changes over time (concern (i)). In the scenario, transportation of food products requires a high number of nodes in the container to be configured for temperature registration. Afterwards, when the same container is used for transportation of electronics, the WSAN has to be reconfigured in order to save on resources. It is unrealistic, or even impossible, to anticipate the needs of all future usage scenarios at development time. Hence, on-the-fly reconfiguration of functionality is an essential feature WSAN middleware should provide. The set of reconfiguration facilities must include operations for dynamic deployment of individual application components and reconfiguration of existing functionality. Secondly, many concerns in WSAN applications are orthogonal to the application. Non-functional requirements such as energy, security, or persistence settings may cross-cut the application. To avoid tangled code, the logic encapsulating these concerns should be separated from code encapsulating functional requirements. In correspondence with evolution of functional requirements, the set of non-functional requirements is also subject to change.

In the logistics scenario, the HeatingController may only react to `AdjustHeating` events sent by authenticated parties. However, containers are usually transported across different locations by subcontracted logistics providers. As such, the set of authenticated parties (i.e. truck and storage companies) is variable over time, as well as the credentials that are applied for securing the system.

2.2 Complex Component Interactions

Events-based middleware provides a good fit for programming highly dynamic environments (concern (ii)). Unfortunately, programming rich inter-node interactions using events frequently results in reasonable amounts of control flow statements to be embedded inside a program (concern (iii)). Consider implementing the TransportMonitor node of the scenario using event-based middleware. Typically, such middleware invokes an *event handler* for each event the node receives. This particular node has to react differently to each received event. Its event handler will therefore have to *dispatch* over each received event. It will also have to *store* events in memory and *match* them with previously received events. For instance, to send an `AdjustHeating` event to the actual container from which an unsuitable `TemperatureReading` event originated, the TransportMonitor must match the address of the event's origin with the payload of a stored `Online` event.

Without adequate support for the aforementioned functionality, developers resort to ad-hoc implementations in event handlers. These are error-prone and bound to be duplicated over the event handlers of multiple nodes. Moreover, event handlers cannot be composed in a straightforward manner. The fact that events carry sensor readings only aggravates this problem for WSANs. Readings of different sensors *expire* at different rates. *Subsumption* of readings might even be deployment-specific. For instance, a fresh sensor reading from a node invalidates stored readings from the other nodes in the same container. Previous reactions to expired or subsumed readings might even have to be *compensated* for. For instance, to avoid overheating when the *HeatingController* gets disconnected from the WSAN.

Even with adequate middleware support for event *dispatching*, *storing* and *matching*, implementing event *expiration*, *subsumption* and *compensation* for WSAN applications still involves a fair amount of bookkeeping. There is therefore a need for comprehensive language support that minimizes the accidental complexity that is inherent to programming WSAN applications using event-based middleware.

3. MIDDLEWARE SOLUTIONS

Figure 2 shows an overview of our integrated middleware. The three main elements are LooCI, an event-based component model for developing adaptive WSAN applications; PMA, an adaptive policy system for enforcement of non-functional concerns; and CrimeSPOT, a domain specific language to implement complex application interactions. Each of these elements is individually needed for dealing with very specific concerns of WSAN applications. First, LooCI and PMA provide support for dynamic adaptation of functional and non-functional requirements (concern (i)). Second, all components communicate with each other over a distributed event-bus making them robust for intermitted disconnections (concern (ii)). Third, developers can program complex node interactions through declarative rules rather than low-level events (concern (iii)).

3.1 LooCI: Event-based Component Model

The Loosely-coupled Component Infrastructure (LooCI) [6] is a lightweight, run-time reconfigurable component model featuring an event-based style of component interaction. All LooCI components define their provided interfaces as the set of events that they publish, whereas the required interfaces of a LooCI component are defined as the events to which the component subscribes. Components are indirectly bound over an event bus abstraction, implementing a decentralized publish/subscribe interaction model (cf. Figure 2).

As result, all communication between LooCI components is carried by semantically typed events that allow for asynchronous and indirect communication between a pair of components. By decoupling communicating components through an asynchronous event-based interaction model, LooCI introduces a loosely-coupled style of system composition, which is beneficial for adaptation. A per-node reconfiguration engine maintains references to all installed components and exposes a remote configuration interface on the event-bus,

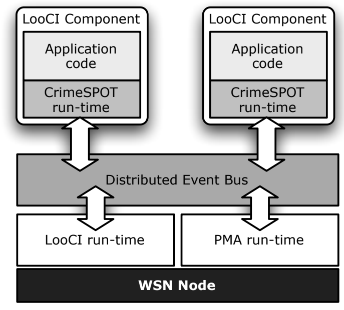

Figure 2: High-level architecture

which allows for reconfiguration and introspection of all local components. The set of run-time reconfiguration facilities in LooCI includes support to dynamically install, remove, start, or stop individual components, in combination with dynamic re-wiring of components over the event bus.

The key benefit of LooCI is that it promotes this event-based interaction paradigm together with components. This aligns well with the dynamic and interrupt-driven WSAN nature. More details about LooCI are provided in [6].

3.2 PMA: Policy-based WSAN Management

The Policy Management Architecture (PMA) [9] offers a policy abstraction to modularize concerns related to configuration and composition of non-functional requirements inside a WSAN application. Policies in PMA are specified using a generic declarative policy language, following Event-Condition-Action semantics.

PMA policies are semantically typed and consist of a description of the triggering events, a condition that is a logical expression typically referring to the triggering events and external system aspects, and a list of actions to be enforced in response. Upon specification, PMA policies are parsed, analyzed, and transformed into a compact byte code representation for efficient dissemination inside the network.

The PMA runtime on every node consists of three key elements: a secure distribution component, a policy repository, and a policy engine. The PMA policy engine is closely integrated with the LooCI event bus (cf. Figure 2). Every incoming and outgoing event that flows over the event bus is first redirected to the policy engine, which checks what policies have to be applied to that event. The distribution component provides operations to dynamically and securely install, enable, disable, or remove a policy. The repository component is used to store all installed policies and retrieve all policies matching a given event. More details on PMA are provided in [9].

3.3 CrimeSPOT: Language Abstractions

CrimeSPOT [3] is a domain-specific programming language to be used on top of event-based WSAN middleware. From a *node-centric perspective*, it enables programming component interactions through declarative rules rather than event handlers. Each rule specifies how the component should react to a particular sequence of events. This relieves developers from having to implement event *dispatching*, *storing* and *matching*. Moreover, component interactions can be composed in a straightforward manner by enumerating the rules that govern them.

CrimeSPOT is domain-specific since it supports associating custom semantics to events that carry sensor readings. Declarations control the aforementioned event *subsumption*, *expiration* and *compensation*. With respect to the latter, 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. From a *network-centric perspective*, CrimeSPOT enables developers to specify which rules are to govern which components. Through macro-programming facilities, the resulting configurations of components and rules are reusable within and across WSAN applications. This network-centric code is compiled to node-level code that is tailored to each individual component.

Key elements of each CrimeSPOT runtime are its inference engine, rule base, and fact base. Storing events in a fact base enables the natural use of pattern matching in rules to relate events through their payload or origin. The inference engine re-evaluates the fact base against the rule base whenever the former changes. The engine is based on a variant of the Rete algorithm [4], an *incremental* forward chainer, that tracks the causality between facts and conclusions. More details on CrimeSPOT are provided in [3].

4. LOGISTICS SCENARIO IMPLEMENTED

We discuss the implementation of the logistics scenario using our comprehensive language and middleware support.

4.1 LooCI Code

Implementing a LooCI component is fairly simple and does not impose major changes in the way how developers write code for the underlying Java ME platform. As such, developers only need to extend a generic base class, providing a number of methods to automatically register the component with the per-node reconfiguration engine and event-bus. The code below illustrates a LooCI component implementing HumiditySensor functionality. This component regularly publishes events of type `HUMIDITY_READING`.

```
public class HumiditySensor extends LooCIComponent
                               implements Runnable {
 private HumiditySensor humSensor =
                    SensorBoard.getHumiditySensor();
 public HumiditySensor() {
   super("HumiditySensor", //component type
    new EventType[]{ EventTypes.HUMIDITY_READING },//provided
    new EventType[]{ }); //none required
 }
 //start() and stop() methods omitted

 public void run() {  //inherited from Runnable
   while(true){
     byte[] value = humSensor.getHumidity();
     publish(new Event(EventTypes.HUMIDITY_READING, value));
     Utils.sleep(600000); //sleep 10 minutes
   }
 }
}
```

Dynamic component deployment and reconfiguration is achieved by creating an interactive session with the gateway device attached to the WSAN. Below is a transcript describing the deployment and configuration of the HumiditySensor component. The component is first deployed over the air to a particular node, followed by the assignment of a unique component ID '3' by the reconfiguration engine on that node (line 1-2). After activation (line 3-4), introspective queries can be issued to retrieve information (e.g. component 3 is queried for provided event types – event ‘101’ (HUMIDITY_READING) is provided) (line 5-6). Finally, the HUMIDITY_READING interface is wired to the TransportController component (ID=6) (line 7-8).

```
1 deploy HumiditySensor.jar 0014.4F01.0000.55E5
2 => 3
3 activate 3 0014.4F01.0000.55E5
4 => true
5 getInterfaces 3 0014.4F01.0000.55E5
6 => [101]
7 wireTo 101 3 0014.4F01.0000.55E5 101 6 0014.4F01.0000.53A2
8 => true
```

4.2 PMA Code

The PMA policy specified below is used to integrate security inside the logistics application of Section 2. Upon policy deployment, some security initialization statements are first executed (line 2-3). Before an ADJUST_HEATING event is delivered to the HeatingController, the event is first checked for authenticity (line 7). When the sender's authenticity cannot be verified, the event is dropped by policy engine (line 9).

```
 1 policy "example" "Auth AdjustHeating events" "AUTH" {
 2    uint8_t cid = 0; //local variable
 3    on INSTALL { cid <- initialize("SHA-1","SHARED_KEY"); }
 4    on UNINSTALL { clear(cid); }
 5
 6    on event ADJUST_HEATING as e;
 7    if( !verify_auth(e,cid) )
 8    then(
 9        deny e;
10    )
11 }
```

4.3 CrimeSPOT Code

Figure 3 depicts an extract of the CrimeSPOT code for the logistics scenario. As it determines which rules govern the interactions of which components, it is network-centric.

Quantified Blocks The network-centric code consists of multiple quantified blocks. These are demarcated by curly braces and preceded by a quantifier, which specifies the component for which the code is intended. Two kinds of blocks can be distinguished. The first kind groups the component's interaction rules. For instance, the block on lines 5–9 encompasses an interaction rule for the TemperatureSensor component. The second kind groups code that implements application logic. They are similar to the other blocks, except that their quantifiers are suffixed with .java.

To enable code reuse within and among WSAN applications, code blocks can be shared by components. Such blocks are quantified either by a component enumeration or by the *-wildcard. The latter is the case for the block on lines 1–3, which defines a macro variable. Macro variables, prefixed by a $-sign, substitute for a textual value at compile-time. They facilitate reconfiguring WSAN applications. For instance, the $publicationInterval macro variable determines both publishing intervals (line 8) and expiration rates throughout the application (lines 7 and 38).

Interaction Rules In general, an interaction rule (e.g., lines 37–44) has a fact in its head (e.g., line 37) and a conjunction of conditions in its body (e.g., lines 39–44). Each condition has to be satisfied in order for the rule to be *activated*. For a condition to be satisfied, there has to exist a fact (i.e., a stored event) that matches the condition under a variable substitution. The bindings for each variable occurrence 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. Whenever a rule becomes activated, the inference engine of the component adds the rule's head either to the local or to a remote fact base. This is determined entirely by meta-data (i.e., everything between @[...] on line 39).

Relating Facts Consider the rule on lines 37–44. As indicated by the enclosing quantifier, it governs the interactions of the TransportMonitor component. The rule adds an appropriate adjustHeating fact to the fact base of the HeatingController in a particular container. The actual ?heatingLevel is computed from a temperature reading ?t and a humidity reading ?h, which get bound through the conditions on line 39 and line 41 respectively. These conditions also bind variables ?tm,?ti, ?hm and ?hi to the originating components' MAC address and component ID respectively. Through a common variable ?c, the conditions on lines 40 and 42 subsequently ensure that these components reside in the same container. Analogously, the condition on line 44 provides bindings for ?cm and ?ci. These are used in the head of the rule to express that the adjustHeating fact should be added to the correct fact base (line 37). Note that the online facts don't have to be received from the network, but could be pre-defined as illustrated by lines 26–29.

Application Logic CrimeSPOT supports invoking application logic from within rule bodies. To this end, it provides conditions of the form "<variable> is <invocation>" which bind their left-hand side to the result of the invocation on their right-hand side. For instance, line 43 binds ?heatingLevel to the result of computeHeatingLevel(). The rule on lines 6–8 schedules its invocations of getTemperature() at an interval of 600 seconds. 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., lines 14–15), the value of which will be sent a message activated(CSVariableBindings) upon activation. Application logic can be implemented in the corresponding method (e.g., to adjust the container's heater).

Fact Subsumption Declarations of the form "incoming <newfact> subsumes <oldfact>" control subsumption. For instance, lines 31–32 specify that a newly received temperatureReading subsumes older ones that were received from the same component. The subsumed facts will be removed from the fact base. To this end, variables ?m and ?i substitute for the MAC address and component ID, respectively, of the component that published the new fact (cf. the from meta-data on line 31) as well as for the MAC-address and component ID of the component that published the older fact (cf. the from meta-data on line 32).

Compensating Actions Finally, 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 or was subsumed. The inference engine will undo the reaction 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., to reset the container's heater).

5. INTEGRATED IMPLEMENTATION

Three implementations of the LooCI and PMA middleware have been proposed: one for the Contiki platform run-

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

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

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 */ }
  };
}
```

```
TransportMonitor {
    online(Container=`A, Component=`TemperatureSensor, MAC="x1:x2:x3:x4", ID=1).
    online(Container=`A, Component=`HumiditySensor, MAC="y1:y2:y3:y4", ID=1).
    online(Container=`A, Component=`HeatingController, MAC="z1:z2:z3:z4", ID=1).
    // …

    incoming temperatureReading(Celsius=?new)@[from(MAC=?m,ID=?i)]
    subsumes temperatureReading(Celsius=?old)@[from(MAC=?m,ID=?i)].

    incoming humidityReading(Percent=?new)@[from(MAC=?m,ID=?i)]
    subsumes humidityReading(Percent=?old)@[from(MAC=?m,ID=?i)].

    adjustHeating(Level=?heatingLevel)@[to(MAC=?cm,ID=?ci),
                                        factExpires($readingPublicationInterval)]
      <- temperatureReading(Celsius=?t)@[from(MAC=?tm,ID=?ti)],
         online(Container=?c, MAC=?tm, ID=?ti),
         humidityReading(Percent=?h)@[from(MAC=?hm,ID=?hi)],
         online(Container=?c, MAC=?hm, ID=?hi),
         ?heatingLevel is this.computeHeatingLevel((Number)?t,(Fraction)?h),
         online(Container=?c, Component=`HeatingController, MAC=?cm, ID=?ci).
}

TransportMonitor.java {
   public Number computeHeatingLevel(Number temp, Fraction humid) { return ...; }
}
```

Figure 3: CrimeSPOT code for the components in the logistics scenario.

ning on 8-bit motes, one for the Squawk VM running on 32-bit SunSPOT motes and one for the OSGi platform running on a variety of mobile devices. All of them share an event bus abstraction over which events are exchanged in the same binary format. This way, the middleware spans across a broad spectrum of contemporary WSAN devices.

We ported the CrimeSPOT runtime, prototyped in Java, to the LooCI implementation for the Squawk VM. The resulting runtime resides within each LooCI component of which the interactions are to be governed by CrimeSPOT. The event handler of such a component forwards the CrimeSPOT facts it receives to the CrimeSPOT runtime. In order to exchange CrimeSPOT facts over the LooCI event bus, we encode them as LooCI events that carry as their payload a textual representation of the fact and a unique identifier. The latter are key to removing facts from remote fact bases in response to fact subsumption, expiration or compensation (cf. Section 3.3).

The predefined fact-to-event encoding suffices for WSAN applications that consist entirely out of CrimeSPOT-enabled LooCI components. For WSAN applications that also feature plain LooCI components, for instance running on one of the smaller supported motes, developers have to specify a bidirectional mapping from plain LooCI events to CrimeSPOT facts. The following declaration maps the LooCI events published by the HumiditySensor component with an integer payload (cf. Section 4.1) to `humidityReading` facts with a `Percent` attribute. This way, the CrimeSPOT-enabled TransportMonitor component (cf. Section 4.3) can react to these events.

```
<=> humidityReading(Percent=?h)@[factExpires($publicationInterval)]
    event(EventTypes.HUMIDITY_READING)@[payload(Integer=?h)].
```

6. EVALUATION AND EXPERIENCES

We have evaluated the performance of our integrated middleware on Java ME CLDC 1.1 compliant SunSPOT motes [10] (180 MHz ARM9 CPU, 512 kB RAM, 4 MB flash, Squawk VM version 'RED-100104').

	ROM	RAM
LooCI runtime	52 kB	37 kB
LooCI component (average)	1.8 kB	26 kB
PMA runtime	28 kB	1 kB
PMA policy (average)	70 bytes	280 bytes
CrimeSPOT runtime (per component)	460 kB	±30 kB

Table 1: Memory footprint

Memory footprint: Table 1 reports on memory usage. The LooCI runtime consumes 52 kB of ROM (0,6% of available flash memory on the SunSPOT). Dynamic memory requirements (RAM) for the LooCI runtime are 37 kB (7% of available RAM). LooCI components consume on average 2 kB of ROM and 26 kB of RAM. The disparity between ROM and RAM requirements of a component can be explained by SunSPOT-specific overhead. The PMA runtime is equally small in terms of ROM and RAM. The CrimeSPOT runtime requires about 460 kB of ROM per component (i.e., 9.7% of available flash memory). However, we made no conscious effort to reduce this footprint at all. As such, sufficient room for improvement exists. The amount of RAM consumed by a rule depends on the complexity of the corresponding RETE network [4]. A rule representing the worst-case situation for 6 conditions consumes about 30 kB of RAM. In addition, every asserted fact consumes about 3 kB of RAM. We attribute these numbers to our completely object-oriented implementation of the RETE network.

Performance overhead: The performance overhead of event publishing in LooCI and processing by PMA is relatively low. It takes 0.5 ms on average to publish an event and redirect it to the policy engine, whereas it requires another 0.5 ms per policy evaluation. CrimeSPOT requires 80 ms on average for a received fact to be added to the local fact base. It then takes another 140 ms before the aforementioned worst-case rule is activated in reaction to this fact. Again, the performance of the runtime is difficult to measure as it depends on the complexity of its RETE network.

Development overhead: It would be challenging to implement the logistics scenario without any middleware support —in particular its dynamic reconfiguration. This is less clear for the domain-specific language support of our solution. An implementation on top of event-based middleware is certainly feasible, but would require implementing event *dispatching*, *storage* and *matching* (cf. Section 2). Not only are ad-hoc implementations error-prone, they are bound to be duplicated across the event handlers of several nodes. These have been shown to violate important software engineering principles such as composability and separation of concerns [7]. Moreover, a significant amount of bookkeeping would be required to implement the scenario's required event *expiration*, *subsumption* and *compensation*. We therefore argue that our domain-specific language support minimizes the accidental complexity that is inherent to programming WSAN applications using event-based middleware.

In conclusion, the processing capabilities of the SunSPOT nodes are more than adequate. However, they are situated at the high-end of the WSAN market. In this regard, we firmly believe that the benefits brought by LooCI, PMA, and CrimeSPOT will outweigh the cost of such nodes as the complexity of WSAN applications increases.

7. RELATED WORK

In recent years, a number of run-time reconfigurable component models have been introduced in the WSAN field, including OpenCOM [5] and RUNES [2]. OpenCOM is a general purpose component model, which has been deployed in a number of real world WSAN scenarios. OpenCOM supports dynamic reconfiguration via a compact runtime kernel. The RUNES [2] model brings OpenCOM functionality to even more constrained devices. Compared to LooCI, OpenCOM and RUNES only support local component interactions and thus adopt a tightly coupled interaction style. Both models do not consider distributed interactions a priori, but realize distribution through custom overlay frameworks. Marsh et al. [8] define a memory efficient policy language solely focussing on the domain of WSAN security. Finger [12] is a lightweight TinyOS-based policy system. Compared to PMA, Finger offers no support of run-time addition of policies containing new events and actions.

FACTS [11] comes close to CrimeSPOT's rule-based approach to programming inter-node interactions. Nodes interact by exchanging facts, which can be reacted to through declarative rules. However, logic variables cannot be used within these rules. Hence, a node cannot react to several related facts. As causality between rule bodies and heads is not tracked, rule deactivation cannot be reacted to. CrimeSPOT's macroprogramming facilities are comparable to those introduced by ATaG [1]. ATaG specifies a WSAN application in terms of tasks that have to be instantiated on particular nodes. Its graphical notation is more expressive concerning the instantiation of tasks on nodes and the interactions between tasks. However, no support is provided for programming the tasks themselves. Reactions to incoming data still have to be implemented through event handlers.

8. CONCLUSION AND FUTURE WORK

We discussed the integration of LooCI, an event-based component model that embraces dynamic reconfiguration; PMA, an adaptive policy system for enforcement of non-functional concerns; and CrimeSPOT, a domain-specific language for implementing complex component interactions. Together, they provide comprehensive support for the complexities that are inherent to developing active WSAN applications. We implemented and evaluated a prototype integration using a scenario from the logistics domain that motivates the need for such support. Although the resulting WSAN application is dynamic in its functional and non-functional concerns, its inter-node interactions were relatively straightforward to implement. In future work, we will further extend the integration of LooCI, PMA and CrimeSPOT. Among others, we will investigate how LooCI components can safely share a single CrimeSPOT runtime when they are deployed on the same node – even if their colocation is short-lived. Another avenue comprises middleware and language support for interacting with the physical n-hop neighborhood of a node.

Acknowledgements: This research is partially funded by the Interuniversity Attraction Poles Programme of the Belgian State of the Belgian Science Policy, the Flemish agency for Innovation by Science and Technology (IWT Vlaanderen), and the Research Fund K.U.Leuven. It is conducted in the context of the IWT-SBO-STADiUM project No. 80037 and IWT-SBO-SymbioNets project No. 090062.

9. REFERENCES

[1] 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 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, 2005.

[2] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2):149–162, 2007.

[3] C. De Roover, C. Scholliers, W. Amerijckx, T. D'Hondt, and W. De Meuter. Language support for programming interactions among wireless sensor network nodes. In *Proceedings of the 5th Int'l Symposium on Ubiquitous Computing and Ambient Intelligence (UCAmI 2011)*, 2011.

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

[5] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes. Dynamic reconfiguration in sensor middleware. In *Proc. of the 1st workshop on Middleware for sensor networks*, pages 1–6, 2006.

[6] D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, W. Joosen, and J. Ueyama. Building wireless sensor network applications with LooCI. *Int'l Journal of Mobile Computing and Multimedia Communications*, 2(4):38–64, October 2010.

[7] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, EPFL, 2010.

[8] D. Marsh, R. Baldwin, B. Mullins, R. Mills, and M. Grimaila. A security policy language for wireless sensor networks. *Journal of Systems and Software*, 82(1):101–111, 2009.

[9] N. Matthys, C. Huygens, D. Hughes, J. Ueyama, S. Michiels, and W. Joosen. Policy-driven tailoring of sensor networks. In *Proceedings of the 2nd International conference on Sensor Systems and Software*, volume 51. Springer, December 2010.

[10] 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*, pages 78–88, 2006.

[11] K. Terfloth, G. Wittenburg, and J. Schiller. Rule-oriented programming for wireless sensor networks. In *Proceedings of EAWMS, DCOSS*, 2006.

[12] Y. Zhu, S. Keoh, M. Sloman, E. Lupu, N. Dulay, and N. Pryce. Finger: An Efficient Policy System for Body Sensor Networks. In *Proceedings of the 5th IEEE MASS*, September 2008.