ESCAPE: A Component-Based Policy Framework for Sense and React Applications

Giovanni Russello, Leonardo Mostarda, and Naranker Dulay

Imperial College London, London SW7 2RH, United Kingdom
{russello,lmostard,n.dulay}@imperial.ac.uk

Abstract. Sense-and-react applications are characterised by the fact that actuators are able to react to data collected by sensors and change the monitored environment. With the introduction of nodes sporting actuators, Wireless Sensor Networks (WSNs) are being used for realising such applications. Sensor and actuator nodes are capable of interact locally. As a result, the logic that coordinates the activities of the different nodes towards a common goals has to be embedded in the network itself. In this scenario, the development of applications becomes more complex.

In this paper, we present a component-based framework that facilitates the development of sense-and-react applications promoting reuse of code. While applications components are used to implement basic functionalities (sense and reaction) our framework allows the specification of application-domain requirements. Our framework is composed of a Publish/Subscribe Broker, a component-based service layer and a Policy Manager. The broker manages subscriptions information and the service layer provides mechanisms orthogonal to publish/subscribe core (e.g., diffusion protocols, data communication protocols, data encryption, etc.). The novelty of our approach is the introduction of the Policy Manager where policies are enforced. Policies are rules that govern the choices and behaviour of the system. They can be used for specifying which services have to be associated with the broker operations. Moreover, policies can embed rules for coordinating the activities of the different sensors and actuators for reaching the common goals of applications.

1 Introduction

Early applications for WSN focused mainly on sensing the environment and sending the data to central sink devices with more computational power (e.g., PDAs and laptops) and therefore were able to coordinate the activities in the controlled environment. More recently, with the development of sensor nodes with more computational power and actuator nodes the *sense-and-react* application paradigm has emerged [1]. Sense-and-react applications are characterised by the fact that the data gathered by the sensing nodes can be used directly by actuator nodes that can react and change the sensed environment. Developing sense-and-react applications for WSNs is complicated by more complex interactions and the stringent limitations that characterise the devices where they are deployed. As a result, often applications developed for this scenario

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 212–229, 2008.

require ad hoc solutions optimised for specific environments. This compromises the flexibility, maintainability, and reusability of such applications [2,3].

Component-based software engineering (CBSE) can play a crucial role as it can be used for balancing the need of reusability with that of providing an efficient programming abstraction [4]. *Application components* encapsulate the functionality of the nodes. Components deployed on sensor nodes can be programmed for sensing data from the environment while components deployed on actuator nodes gather the data and react accordingly. However, to fully take advantage of CBSE it is necessary to provide a layer of abstraction to glue together the functionality of the different components. Components need to coordinate their functionality to achieve the global goals of the system without having their code tangled with details concerning OS and networking services. An effective solution to this problem needs to offer an appropriate level of abstraction to components without being too demanding in terms of resources. With its loosely-coupled, event-driven messaging services, the publish/subscribe paradigm offers to applications simple yet powerful primitives for communication.

Although the publish/subscribe paradigm is quite widely used, different aspects of its model concerning notification distribution, delivery and security can be implemented using different mechanisms. Each mechanism imposes requirements in terms of resources and each is more suited for specific class of applications. For instance, for certain critical applications it is acceptable to use a reliable delivery protocol even if it requires more resources in terms of energy consumption and computational overhead. Such aspects are not directly related to the basic functionality of the application and as such they can be referred to as *extra-functional concerns*. The *Separation of Concerns* (SoC) principle advocates that the basic functionality of an application should be specified in isolation from details regarding extra-functional concerns [5,6]. Because application code is not tangled with other details regarding extra-functional concerns, the code that results is less prone to errors, easier to maintain and far more reusable.

In this paper, we propose a component-based framework for programming WSNs realised through the publish/subscribe paradigm where extra-functional concerns are encapsulated in middleware components. Application developers specify which mechanisms have to be used in their applications in terms of *policies*. Policies are rules that govern the behaviour of a system and are an effective solution for separating the application functionality from low-level mechanisms [7]. Our framework supports an Event-State-Condition-Action Policy Environment (ESCAPE) where policies *connect* components orthogonally to the publish/subscribe paradigm. Policies define *stateful* interactions among components to *coordinate* their activities and reach system-wide goals.

The contributions of this paper are the followings. Our framework realises a flexible publish/subscribe system inasmuch as the needs of different application scenarios can be catered for by the different mechanisms that it can support. Application developers use policies to define which of these mechanisms are to be used. Because policies are defined outside the application code, application components (encapsulating application functionality) and middleware components

(encapsulating mechanism implementations) become the unit of reusability that can be deployed without modification in different scenarios. Moreover, policies represent also a unit of reusability. Once a specific behaviour is defined in a policy that policy can be deployed in other applications with similar characteristics. Finally, our framework is extendible since new mechanisms can be implemented as middleware components and deployed on the nodes.

The rest of this paper is organised as follows. In Section 2, we discuss the motivations and requirements of our approach. Section 3 provides a description of the architecture of our framework. Policy syntax and semantics are described in Section 4. To validate our approach, we present in Section 5 a case study and some of the policies used for its realisation. A brief evaluation of the implementation of our framework is presented in Section 6. In Section 7, we compare our approach to related research. We conclude by highlighting some future research direction in Section 8.

2 Motivations and Requirements

Sense-and-react applications represent a class of embedded control systems characterised by the realisation of a feedback-loop between a *sensing apparatus* and a *reacting apparatus*. Some examples of sense-and-react applications are heating, ventilation, and air conditioning (HVAC) [1], fire alarm systems, and burglar alarm systems. Nodes capable of sensing the environment provide readings of some parameters forming the sensing apparatus. Nodes equipped with actuators react to specific events and change the environment according to user preferences. As a software system, a sense-and-react application consists of two parts: the main *functionality* and the *control laws*. The main functionality represents the basic logic that is mapped into application components deployed in each sensor node. For instance, an application component deployed on a temperature node provides the functionality to obtain the readings from the node hardware and make it available to other nodes in the WSN. The control laws map the sensed data to specific actions that should comply with user preferences. When the functionality and control laws are not intertwined then it becomes possible to share the functionality of a node among different applications controlling the same environment. For instance, the functionality of a temperature node could be shared by both a HVAC and a fire emergency application. The two applications have different control laws however the functionality of the temperature node for both applications is the same, e.g. providing readings for the temperature of the environment.

From the above simple example, it emerges that the development of sense-and-react applications for WSNs is challenging not only for the constrains imposed by the physical devices but also for the complexity of the interactions that can be realised among different applications. In the following, we try to identify a set of requirements to define key aspects for facilitating the development of such applications.

Minimise functionality to maximise reuse. In our framework, the node functionality is encoded as an application component deployed on the node. In order

to maximise the reuse of such functionality, component functionality should be agnostic of the control laws enforced in the environment.

Coordination through middleware. WSNs can be seen as miniaturised distributed systems. The publish/subscribe model offers a very powerful abstraction for realising loosed-coupled distributed applications and middleware implementations have been already proposed in WSNs. In particular, the *reactive* style of interaction makes the model attractive for sense-and-react applications. Notifications sent by the sensing nodes can be used for trigger reactions by actuators. The underlying middleware is also the ideal place where the SoC principle should be realised. In particular, the middleware should provide to the application developers mechanisms that would allow the selection of different strategies implementing extra-functional concerns that can be subsequently enforced at runtime. For example, if a particular notification strategy is required, the middleware should offer such a strategy implemented as a component. Application developers specify which particular components have to be used with their applications in terms of policies. If necessary, new mechanisms can be developed and deployed as well, independent of the application functionality implemented by components.

Stateful policies. The control laws in sense-and-react applications typically represent *transitions* through different *states*. For instance, a sprinkler node that receives a temperature reading higher than a certain threshold has to check that smoke is detected before opening the water. This behaviour can be represented as two transitions: (i) from a normal state to a pre-alarm state when the temperature is above a safety threshold; and (ii) from a per-alarm state to an alarm state when the smoke detector provides a positive reading. To increase functionality reuse, control laws should be specified in isolation from application components. Policies represent ideal candidates for specifying the control laws of the system. In this case, policies need to be able to capture states and specify how transitions through different states must be executed.

Localised vs distributed computation. Sense-and-react applications are characterised by their capacity of reacting to stimuli coming from the surrounding environment. Because actuators can be in the proximity to where the data is generated, it is not necessary to flood the network with all readings. However, in certain case it is necessary that events have to be spread through the nodes present in the environment, such as fire alarms.

Multiple interaction patterns. Although reactive interactions characterised sense-and-react applications, there are still cases where *proactive interaction* should be preferred instead. This type of interaction is common in sense-only applications where data is proactively required by the consumers. In this way, it is possible to save the energy of the sensing nodes that are requested to generate the data only when it is needed.

In the following, we describe the architecture of our system to satisfy the identified requirements.

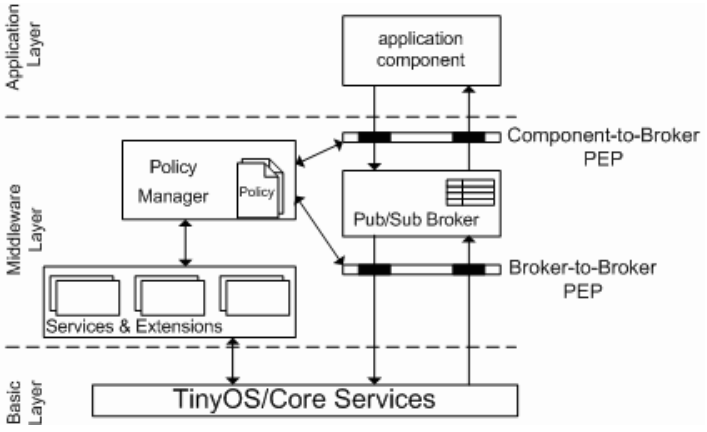

Fig. 1. Overview of our component-based architecture for a WSN node

3 Architecture

In this section, we discuss the main features of our approach. As shown in Figure 1, our approach presents a layered structure composed of an *application layer*, a *middleware layer* and a *basic layer*.

The application layer contains the basic functionality that is deployed on each sensor node. The functionality provided by a sensor node is encapsulated in application components. For instance, an application component deployed on a temperature sensor is responsible for providing temperature readings and acts as a publisher for this type of notification. Components deployed on actuator nodes are responsible for controlling and activating the actuator hardware according to the actual needs. In this case, actuator components act as subscribers of notifications representing the actual conditions of the environment.

The middleware layer consists of a **Publish/Subscribe Broker**, a **Policy Manager**, and a **Services & Extensions** described as follows.

Publish/Subscribe Broker. This module provides an API to the application layer and manages the subscription tables. In our approach, a notification is a tuple of (attribute,value)-pairs. A subscriber specifies its interest in a notification by issuing a `subscribe(notification)`. Although the subscriber cannot directly express constraints on the content of notification using the API, constraints can still be expressed in policies. For instance, if a subscriber should be notified only if the temperature value is higher than 50, then it is possible to write a policy that inspects the values of the temperature notifications and discards the notifications with values lower than 50 (more on this in Section 5). This decoupling of component functionality from subscription constraints increases the reusability of the components without sacrificing the expressivity of the publish/subscribe abstraction. In our framework content constraints are used for expressing control laws in the form of policies. A publisher advertises its notifications using

`advertise(notification)` and it publishes the data using `notify(notification)`. Subscription and advertisements can be withdraw using `unsubscribe(notification)` and `unadvertise(notification)`, respectively.

Policy Manager. One of the main features of our framework is that the publish/subscribe core is decoupled from mechanisms related to notification delivery, subscription distribution, and communication protocols. This design decision increases the flexibility of our approach inasmuch as our middleware is not bound to any specific mechanisms. Application developers can select the appropriate mechanisms that suit best their application needs. In contrast to the approach presented by Hauer et al. [8] where application components have to explicitly specify the mechanism to be used, in our framework components are completely agnostic of such specifications. Instead, we propose a policy-based approach where policies are used for such specifications. The enforcement of policies is done by a Policy Manager module. Policies can be specified to be enforced at specific points in our framework. Component-to-broker and broker-to-broker interactions are monitored via Policy Enforcement Points (PEP). Each time a message is sent through these interaction channels, the corresponding PEP intercepts the message and sends an event to the Policy Manager. The Policy Manager uses the events to trigger the policies available in its repository defined for that PEP (more on how policies are specified and enforced in Section 4). An important feature of our policy environment is that, the Policy Manager supports the deployment of new policies even during run-time without the need of taking the running application off-line. This feature increases flexibility of our framework and it makes particular appealing for WSN applications that required a high degree of availability.

Services & Extensions. This module provides hooks to the policy environment for invoking components that implement protocols and services outside the publish/subscribe core. Each service component is responsible for providing and consuming information required for fulfilling their tasks and if necessary to perform specific actions. Figure 2 shows some of the services components currently available. Components are organised in two sets: *Basic Pub/sub Components* and *Extension Components*. The Basic Pub/sub Components provide services that are necessary for realising the publish/subscribe paradigm and are described as follows:

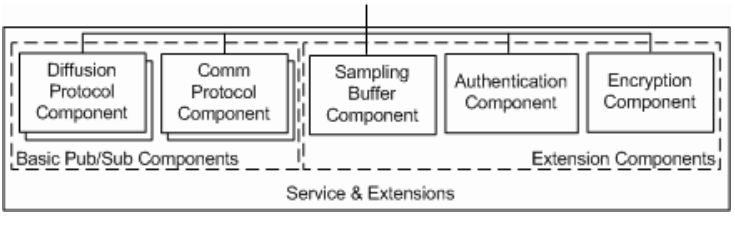

Fig. 2. A view of the Service & Extensions module

- A *Diffusion Protocol Component* (DPC) is responsible for routing data between publishers and subscribers. Initial work on diffusion protocols used a *two-phase pull* model [9], where subscriptions are distributed for the seeking of matching advertisements. Once a matching advertisement is found, the notifications are sent to the subscribers trying to find the best possible paths. This type of protocol is not suitable for all classes of application. In applications with many publishers that produce data only occasionally, the two-phase pull model is inefficient since it generates a lot of control traffic for keeping the delivery route updates. For this class of applications, the *push* diffusion protocol was proposed in [10]. According to the push diffusion protocol, the subscriptions are kept locally and the notifications seek subscribers. Other diffusion protocols have also be proposed, such as a *one-phase pull* protocol [10] (an optimised version of the two-phase pull), geographically scoped protocols [11], and rendezvous-based protocols [12,13]. Our current implementation provides a Push DCP (PsDCP) and a Pull DCP (PlDCP) implementing the push and one-phase pull protocols, respectively.
- A *Comm Protocol Component* (CPC) implements the delivery protocols of the messages generated by the publishers and subscribers with certain delivery guarantees. For instance, in certain cases subscriptions need to be updated frequently then a CPC that implements a probabilistic communication algorithm is acceptable. On the other hand, if an application requires a more reliable subscription distribution then a CPC that offers an algorithm that performs control traffic communication in the background can be used (at a higher costs in terms of resources). The former protocol is implemented by Probabilistic CPC (PCPC), while the latter is implemented by Reliable CPC (RCPC).

The Extension Components provide extra features to the paradigm. In the following we describe the components that have been implemented for the realisation of our case study discussed in Section 5.

- The *Sampling Buffer Component* (SBF) provides functionality for storing data samplings and computing certain predicates on the stored values. For instance, it could be used for calculating if a recent sampling differs more than a specified delta value from a stored sampling. Alternatively, it could just be used as a buffer that stores samplings frequently accessed or that requires a long time to be collected (i.e., audio signals).
- The *Authentication Component* and the *Encryption Component* are used for implementing *Trust Groups* of sensors. Trust groups are similar to secure multicasting groups [14]. Each trust group is associated to a secret key K_g. In this way, members of the same trust group are able to perform encryption and authentication within the group.The distribution of the K_g is done as an out-of-band bootstrapping process. For the encryption/decryption the component uses the Skipjack algorithm provided by the TinyOS core.

4 Event-State-Condition-Action Policy Environment (ESCAPE)

In this section we define the syntax and the semantic of our policy language. The syntax is defined by using a Backus-Naur form (BNF) while the semantic is described by defining the run-time behaviour of our policy manager.

```
<Policy> = policyName policyVariables <ESCAlist>

<ECSAlist> = "on" <Event> <SCAlist> | "on" <Event> <SCAlist> <ECSAlist>

<SCAlist> = currentState "-" newState "{"condition"}" "->" "{"action"}" <SCAlist>
 | currentState "-" newState "{"condition"}" "->" "{"action"}"

<Event> = <Qualifier> <pubSubEvent> | timeout <Timeout>

<Qualifier> = "B-C" | "C-B" | "B-extB" | "extB-B"

<pubSubEvent> = "notify(T)" | "advertise(T)" | "unadvertise(T)" | "unnotify(T)" |
"unpublish(T)" | "unsubscribe(T)"
```

Fig. 3. The syntax to our language for specifying ESCA policies

In Figure 3, we show our grammar used to define Event-State-Condition-Action (ESCA) policies. The notation $< symbol >$ is used define a non-terminal symbol, the character " is used to enclose language's keywords while words are terminals (i.e., symbols that never appear on the left side of a definition).

A policy is composed of a *policyName* followed by a *policyVariables* and an *ESCAlist*. The terminal *policyVariables* denotes variable declarations that can be used inside *condition* and *action* definitions. An *ESCAlist* is a list of event-state-condition-action each starting with the keyword **on** followed by an event and its state condition action list (i.e., SCAlist). A state-condition-action (SCA) is of the form *currentState-newState condition* $- >$ *action* where: (i) *currentState* is an integer that denotes the current policy state; (ii) *condition* is a predicate that must be true in order to apply the action *action*; (iii) *newState* is the new policy state after the action application.

In our approach, policies can be enforced when pub/sub operations are executed. Each of these operations is associated with a corresponding event of type **pub-sub** defined as following: $notify(T)$, $advertise(T)$, $subscribe(T)$, $unnotify(T)$, $unadvertise(T)$, $unsubscribe(T)$. T is represented by a tuple. With $T.x$ we denote the value of the parameter x in the tuple. For example, a policy defined on an event $subscribe(T)$ will be triggered each time the operation `subscribe(T)` is executed. However, our enforcement mechanism is able to capture the execution of an operation in 4 different points. This means that a pub-sub event associated to an operation can be generated in each of these points. In order to specify at which particular point a policy should be triggered, each pub-sub event must be always preceded by a qualifier that can be: (i) **C-B** specifying that the pub-sub event is sent from the component to its local broker; (ii) **B-C** specifying that the pub-sub event is sent from the local broker to a local component; (iii) **B-extB** specifying that a pub-sub event is sent from the local broker to an external one; (iv) **extB-B** defining an pub-sub event sent by

an external broker to the local one. Characterising a pub-sub event based on its local source and its local destination allows the description of flexible policy specifications. For instance policies can include component-broker interactions in order to filter pub-sub event content while broker-broker interactions allow the introduction of new features (e.g., security) without affecting the middleware basic mechanisms.

We also support **timeout** events for a node. A timeout event is of the form **timeout** t and is executed when for t seconds no event is observed. Generally speaking, a timeout is a way to perform actions when no pub-sub events are observed within a time interval.

Predicates can refer to event parameters, contain policy variables and invoke external libraries. Actions can modify policy variables, modify event parameters, execute any pub/sub operations (e.g., `notify(T)`, `subscribe(B)`, etc.) and call external libraries.

The action specification must always end either with the outcome **accept** or **discard**. Accept (discard) specifies that the pub/sub operation that triggered the policy must be completed (discarded) after the action execution terminates.

4.1 Policy Execution Model

In our model, the enforcement of policies is triggered by events. One event may trigger multiple policies at the same time. In the following, we define our policy execution model, that is the policy manager run-time behaviour. We denote with P the set of all policies and $p_1, \ldots, p_n$ are elements in P. A policy p in P is a set of events $\{e_1, \ldots, e_n\}$. Each event e has related a $SCAlist$ containing a sequence of elements of the form $(cs_i, ns_i, condition_i, action_i)$, representing the current state, the new state (after the action is executed), the condition, and action, respectively. In order to refer to one of these elements we prefix it with the event name followed by the symbol ".". For instance if e is an event then $e.condition_i$ and $e.action_i$ denote the action and the condition related to ith element in the $SCAlist$ of e.

Policies are executed by our policy manager that receives each event e and invokes the `execute` procedure, as shown in Figure 4. The `execute` procedure takes as an input an event e and defines a local list `Outcome` that will contain

```
1  void execute(event e)
2   Outcome[]={}; //the set of outcomes for each executed action
3   if no policy defines e then
4     accept;
5     return;
6   for each policy p that defines e do
7     let CS be the current state of the policy p
8     for i = 0 to p.e.SCAlist size do
9       if (p.e.cs_i == p.CS) and (p.e.condition_i) then
10        execute p.e.action_i;
11        add the outcome of p.e.action_i in Outcome[];
12        break;
13  select an outcome from Outcome[];
```

Fig. 4. Policy execution

the outcomes of all policies triggered by the event e. If no policy is triggered by the event e, then the operation that generated the event e is accepted (line 4) and the procedure terminates. On the other hand, for each policy p that defines e (line 6), then a $SCAlist$ element must be selected. An element in the $SCAlist$ is selected when the current state defined in the element ($p.e.cs_i$) is the same as the current state of the policy (CS) and the condition defined in the element ($p.e.condition$) is satisfied (line 9). In this case, the action corresponding to that element is executed and the outcome statement of the action (either accept or discard) is inserted in the `Outcome` set. When the actions of all policies defining the event e have been executed, the policy manager analyses the `Outcome` set. Because the same event may trigger several policies, the action outcomes that are inserted in the `Outcome` set are in conflict, i.e. both the statements *accept* and *discard* are present in the set. In our approach we allow policy writers to specify which of the two statements must be given priority per event. They can associate with each event e a default statement (either accept or discard) that is executed each time a conflict is detected. This choice allows us to have a fine-grain conflict resolution strategy at runtime that undertakes different statements for different types of events.

4.2 Tool and Policy Analysis

In this section, we describe the process that leads from policy definitions to code generation. This process is implemented by using two separate tools, i.e., an ESCA translator (shown in Figure 5) and the GOANNA tool [15] (shown in Figure 6). The ESCA translator parses each policy and translates it in a state-transition (a state machine). These state machines are input to the GOANNA tool that can show them in a graphical form, performs different semantic checks and generate the policy manager code. In the following we show how the ESCA translator translates a policy and we show the basic components of the GOANNA tool.

A translator checks that each policy p is syntactically correct and produces a state machine A_p. In particular for each event e the translator considers each state-condition-action (e.g, *currentState-newState condition* $- >$ *action*) and adds to A_p a transition that exits from the state *currentState*, is labelled with [*condition*] *e action* and enters in *newState*. In other words when the state machine is in the state current-state, the event e is observed and the condition is true than the state machine can move to *newState*. In Figure 6 we show the state machine related to the following policy:

```
1 TemperaturePolicy
2 on C-B advertise(Temperature,t_value)
3  0-1: true->{accept;}
4 on C-B notify(Temperature,t_value)
5  1-1: {t_value<50} -> {discard;}
```

State machines in this form can be loaded by the GOANNA tool that performs some semantic checks (possibly because of the state machine structure) and generates the policy manager code.

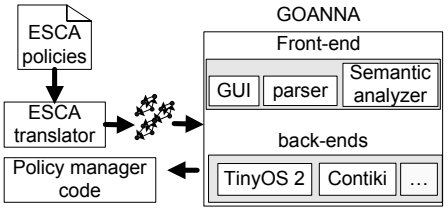

Fig. 5. Process of code generation

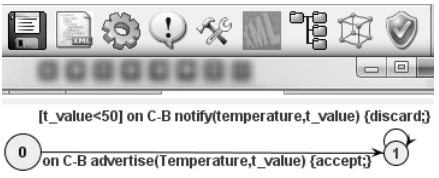

Fig. 6. GOANNA tool

GOANNA uses a front-end and a set of back-ends used for checks and code generation, respectively. The front end is composed of three main components: a *GUI*, a parser and a semantic analyser. The GUI implements a graphical tool to visualise the policy in a graphical form (i.e., a state machine based form). The parser and semantic controller take as input state machines and perform all syntactic and semantic checks, respectively. In the following we introduce the main semantic checks the tool performs, i.e., *state reachability*, *correct sequence* and *recursive event detection*.

State reachability ensures that each event-state-condition-action can be applied, i.e., all states inside a state machine definition can be reached. Correct sequence analyses the state machine definition and verifies correct ordering among system events, e.g., a notify of an event is preceded by a publication. Recursive event detection avoids policies leading to livelock. In the simplest case we can have a policy in which an event e is defined, the new state is equal to the current one (a transition that enters and exits in the same state) and its action define a notify of the same event e. In this case the policy can generate an infinite number of events e without making any progress. Generally speaking a policy can defines a chain of events that produces livelock. For instance a policy can define the events e_1 and e_2 and the state machine is such that: (i) e_1 changes the state from q_1 to q_2 and its action generates an event e_2; (ii) e_2 changes the policy state from q_2 to q_1 and its action generates the event e_1. Our tool tries to visit each state machine, detect possible livelock conditions and produce warnings. As future work we are adapting other well known state machines verifications to our particular context. For instance we are planning to apply checks defined over several state machines by defining composition among state machines and performing checks on it.

5 Case Study

This section presents a case study related to a cultural asset transportation service used to securely move cultural assets from one venue (museum) to another. The service was developed as part of the EU CUSPIS project [16].

In the transportation service, a lorry transports a set of packages each containing a cultural asset. As shown in Figure 7, the lorry is equipped with sensors and actuators on which several sense-and-react applications are deployed.

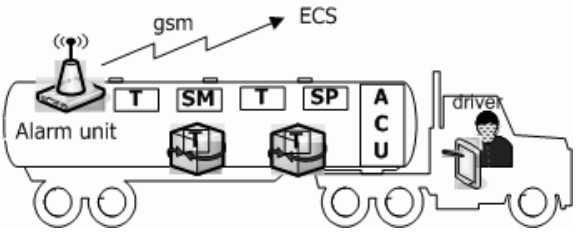

Fig. 7. An overview of the deployment of the sensors and actuators for the CUSPIS case study

During transportation, the lorry is monitored by the Emergency Central Station (ECS) that is in contact with police and emergency units (such as fire fighter stations). To send alarms to the ECS, the lorry is equipped with multiple alarm units that include a GSM transmitter and GPS sensor. The lorry driver can use a portable wireless computer, such as a PDA, to check sensor readings and be notified in case of any alarms.

The following sense-and-react application are deployed:

- *Fire Alarm Application* is responsible for detecting and taking initial actions against the fire inside the lorry. Temperature sensors provide readings for the actual temperature and smoke detectors are used for sensing the presence of smoke. If the temperature rises over a given threshold and the smoke detectors provide positive smoke readings then the water sprinklers must be activated and a fire alarm sent. Issuing a fire alarm activates the alarm unit that informs the lorry driver through the PDA and sends an alarm message to the ECS.
- *Air Conditioning Application* is responsible for maintaining temperature and humidity within the lorry to given values. Temperature sensors (shared with the Fire Alarm Application) and humidity sensors provide reading of the air quality in the lorry. An air conditioning unit (ACU) uses the readings from the sensors to increase or decrease the temperature and humidity to keep those values within the target values set by the driver.
- *Package Tampering Monitor* is responsible for the integrity of the packages containing the artifacts and to raise an alarm in case the packages are tampered with. Each package contains sensors that collect readings for temperature, humidity and light. An indication that a package was opened can be signaled when a reading deviates significantly from the previous values. For instance, when the package is opened the amount of light and temperature inside the package increases and such variation can be captured by the sensor. If this is the case, then the sensor notifies the driver's PDA and the alarm unit. The latter sends an alarm (together with the GPS position) to the ECS to summon the intervention of the police.

In the following, we discuss the policies used for specifying the control laws of the applications. For brevity reasons, we cannot present the complete set of

```
1  global target_t, delta_t;
2  TemperaturePolicy
3    on C-B notify((Temperature,t_value))
4     0-0: {t_value<50 && !SBC.deviateOrZero(target_t, delta_t,t_value)}
5                      -> {discard;}
6    on B-extB notify((Temperature,t_value))
7     0-1: {t_value>50} -> {PsDPC.notify((Temperature,t_value));
8                           RCPC.notify((Temperature,t_value));
9                           accept;}
10   on B-extB notify((Temperature,t_value))
11    0-0:  {SBC.deviateOrZero(target_t, delta_t,t_value)}
12                     -> {PlDPC.notify((Temperature,t_value));
13                     PCPC.notify((Temperature,t_value));}
14                     accept;
```

Fig. 8. The temperature policy defining the control laws for the Fire Alarm and Air Conditioning Applications

policies used for the case study but we have to limit our discussion to the most significant ones. We assume that to avoid the injection in the system of notifications generated from sensors outside the lorry, all the message exchanged use the authentication and encryption components for a trust group communication.

Content-based filtering. The application component deployed on the temperature sensors provides readings of the temperature inside the lorry. Each component acts as a publisher of the notification type `(Temperature,t_value)`. This type of notification is shared by the Fire Alarm and Air Conditioning Applications. Instead of flooding the network with every temperature sampling, only notifications with meaningful values should be allowed to leave the publisher node. In particular, for the Fire Alarm Application, only samplings with values over 50 should be allowed. For the Air Conditioning Application a different approach is used: a notification is published if the difference between the actual value is either more than delta from a target value or it is equal to zero. In the first case, this means that the Air Conditioning Unit (ACU) has to be activated to bring the temperature within the desired target; while in the second case the ACU can be switched off since the desired target is reached. This content-based filtering can be specified by a policy as shown in Figure 8. When the component sends a notification (line 3), the policy checks whether the value of the temperature is less than 50 and that the predicate `deviateOrZero`, provided by the Sampling Buffer Component (SBC), is not satisfied (line 4). In this case the notification is discarded.

However, when the notifications have to be delivered, two different delivery protocols must be used for the two applications. For the Fire Alarm Application, the notification should be spread as quickly and reliably as possible. In this case, the notifications are associated with the diffusion protocol component that implements the push model (PsDPC) using a reliable communication protocol component (RCPC) (line 6-9). On the other hand, for the delivery of the temperature notifications for the Air Conditioning Application (11-14) the pull model with the probabilistic communication protocol is used (implemented by the PlDPC and PCPC, respectively).

```
1  TamperingPolicy
2   on C-B advertise((PackageTemperature,t_value))
3    0-1: true -> {advertise((PackageAlarm));
4                  subscribe((PackageTemperature,t_value));
5                  accept;}
6   on C-B notify((PackageTemperature, t_value))
7    1-1: {!deviateDelta(t_value, delta_value)} -> {discard;}
8    1-2: {deviateDelta(t_value, delta_value)} -> {startTimer(3);
9                                                  accept;}
10  on B-C notify ((PackageTemperature, t_value))
11   2-2: {notification_id == this.node_id} -> {discard; \\ignore: this is my
         notification}
12   2-1: {node_id != this.node_id} -> {discard; \\false alarm: increase in
         temp in other sensors}
13  on timeout()
14   2-3: true -> {notify((PackageAlarm));
15                 discard;}
```

Fig. 9. The policy for setting off alarm notifications when the packages are tampered with

Localised computation. The Package Tampering Monitor is responsible for monitoring the integrity of the packages containing the artifacts and for raising an alarm in case the packages are tampered with. Each package contains a sensor that raises an alarm if the readings for temperature, humidity and light drastically change. For instance, when the package is opened the amount of light and temperature inside the package increases and such changes are can be captured by the sensor. However, care must be taken to avoid notification of false alarms. For instance, if the temperature in the package increases it could be an effect due to the increase of temperature inside the lorry (i.e., the air conditioning unit is not working properly). If this is the case, then the sensors in the other packages also register an increase in temperature. Therefore, before sending the alarm notification, the sensor that first registers an increase of temperature sets off a timer and waits for notifications from the sensors in other packages that signify an increase in temperature. If these notifications from other sensors arrive before the timer timeouts then no alarm is sent. Otherwise, the alarm notification is sent.

This behaviour can be codified using a policy as shown in Figure 9 (note that to improve readability we removed all details related to diffusion and communication protocols). This policy captures only the case for variations in temperature readings. The policy starts registering the node as a publisher for the `PackageAlarm` notification and as subscriber of the `PackageTemperature,t_value` notifications. The publishing of the temperature readings uses a "send-on-delta" approach, where a notification is published only if varies more than a given delta from the previous published notification. The predicate `deviateDelta` is used for checking whether the difference between the actual reading and the previous published one is grater than delta. If the difference in not more than delta, the notification is blocked (line 7). Otherwise, if the notification deviates more than delta, the notification is sent and a 3 second timer is started (line 8). At this stage, the following can happen:

Component type	Description	Code (bytes)	Data (bytes)
Temperature Component	Senses the temperature	4530	40
Broker	Maintains table	1234	21
Policy Manager	Instrumentation code	1120	58
Tampering policy	the policy code	870	12
PsDPC	Push diffusion	680	55
PlDCP	Pull diffusion	730	90

Fig. 10. Code and data size information

- a notification arrives but is the one that was just sent by the node itself (line 11). In this case, the state of the policy is not changed.
- a notification arrives from other nodes (line 12). This means that the increase of temperature is not local to this package but other sensors are registering it as well. Therefore this is a false alarm and should be ignored.
- the timer expires and a timeout event is sent (line 13). This means that no other sensors registered the increase in temperature. In this case a `PackageAlarm` notification is sent (line 14).

6 Implementation

We have used our approach to implement the requirements of our CUSPIS application. We have used the TinyOS operating system running on *Tmote Sky* motes. In particular, we have built basic monitoring components that only sense environmental data (i.e., temperature, smoke, light) and have added our framework to build CUSPIS functionalities.

In Figure 10 we show information about the code / data size of both application components and our framework. We emphasise that the policy manager size is independent from policy specifications and all other components. In other words the policy manager is a container that manages the policy life cycle (i.e., it loads, executes and deletes policies) so that its size is always the same. In our case the temperature components is bigger than the tampering policy since the temperature component must embed all code needed to sense and to manage the timer (the sensing is performed at each tick) while the policy only embeds few if statement and few variables to implement the state machines.

7 Related Work

The TeenyLIME middleware is specifically designed to address the requirements of sense-and-react applications for WSNs. TeenyLIME provides a programming model based on the tuple space paradigm. The tuple spaces in TeenyLIME represents a shared memory that is shared among sensors within a one-hop region. Although the TeenyLIME offers a simple but powerful abstraction, it lacks the flexibility of our approach. In fact, extra-functional mechanisms that are

provided with the middleware are fixed to specific hard-coded modules. For instance, in TeenyLIME tuples are distributed according to a communication protocol that supports only one-hop communications. In our case, notifications can be distributed using several diffusion protocols, according to the needs of the applications.

TinyCOPS [8] is a publish/subscribe middleware that uses a component-based architecture for decoupling the publish/subscribe core from choices regarding communication protocols and subscription and notification delivery mechanisms. The middleware can be extended with components that provide additional services (i.e., caching of notifications, extra routing information, etc.). The specification of which particular mechanism has to be used is done by means of metadata information that the application components have to provide through the publish/subscribe API. In our case, application components are agnostic of such extra-functional concerns since policies are used for specifying which mechanisms have to be used.

The Mires middleware [18] is also a publish/subscribe service that uses the component architecture of TinyOS 1.x. Like our approach, it uses a topic-based naming scheme. However, differently than in Mires, we can support content-based filtering by means of policies. Although in Mires it is possible to introduce new services (like aggregation) using extension components, the choice of the communication protocols is fixed.

MiLAN [19] is a middleware for WSNs that provides application QoS adaptation at run-time. The middleware continuously tracks the application needs and optimises network usage and sensor stacks for an efficient use of the energy. As such, MiLAN focuses more on a class of resource-rich wireless networks that can support well the impact of the monitoring overhead. In our approach, we concentrate more on sensors with limited resources, where optimisations are mainly performed at compile-time.

8 Conclusions and Future Work

In this paper, we have described a component-based framework for WSNs based on publish/subscribe paradigm where ESCA policies can be enforced. Component applications implement the basic functionality of the wireless nodes (sense and reaction capabilities) while policies implement all extra-functionalities that are domain specific. Policies are specified using our state machine language that includes variables and libraries in order to define complex policies. Policies are input in our tool that performs semantic checks and generate all needed code to execute them. We have applied our approach to a case study where a sensor network is deployed in lorries that transport cultural assets between museums. The system has been developed for the TinyOS2 operating system.

Our future work aims to optimise the code generation for TinyOS2 and to carry out a detailed evaluation of the run-time costs. Another direction to explore is to provide to application developers means for selecting the mechanisms that suit best the needs of their applications. Moreover, as the needs of the

applications changes after deployment, an autonomic approach that would select the best mechanisms for the actual needs of the application would be ideal. However, such an approach requires continuously monitoring of the activities that would incur in some overhead. In our future work, we want to study whether the monitoring and adapting overhead is sustainable compared to the overall gain in performance.

Acknowledgments

This research was supported by the UK EPSRC, research grants EP/D076633/1 (UBIVAL) and EP/C537181/1 (CAREGRID). The authors would like to thank our UBIVAL and CAREGRID collaborators and members of the Policy Research Group at Imperial College for their support.

References

1. Deshpande, A., Guestrin, C., Madden, S.: Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28(1) (2005)
2. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Transactions on Networking (TON) 11(1) (2003)
3. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: An acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1) (2005)
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: ASPL 2000. Proc. of the ninth international conference on Architectural support for programming languages and operating systems (2000)
5. Dijkstra, E.W.: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer, Heidelberg (1982)
6. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
7. Sloman, M., Magee, J., Twidle, K., Kramer, J.: An Architecture for Managing Distributed Systems. In: Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 40–46 (1993)
8. Hauer, J., Handziski, V., Kopke, A., Willig, A., Wolisz, A.: A Component Framework for Content-Based Publish/Subscribe in Sensor Networks. Wireless Sensor Networks, pp. 369–385 (2008)
9. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, Boston, MA, USA, August 2000, pp. 56–67. ACM, New York (2000)
10. Heidemann, J., Silva, F., Estrin, D.: Matching data dissemination algorithms to application requirements. In: SenSys 2003. Proc. of the 1st international conference on Embedded networked sensor systems, New York, USA (2003)
11. Yu, Y., Govindan, R., Estrin, D.: Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report TR-01-0023, University of California, Los Angeles, Computer Science Department (2001)

12. Braginsky, D., Estrin, D.: Rumor routing algorithm for sensor networks. In: Proceedings of the First ACM Workshop on Sensor Networks and Applications, Atlanta, GA, USA, October 2002, pp. 22–31. ACM, New York (2002)
13. Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S.: GHT: A geographic hash table for data-centric storage. In: Proceedings of the ACM Workshop on Sensor Networks and Applications, Atlanta, Georgia, USA, September 2002, pp. 78–87. ACM, New York (2002)
14. Rafaeli, S., Hutchison, D.: A Survey of Key Management for Secure Group Communication. ACM Computing Surveys 35(3), 309–329 (2003)
15. Mostarda, L., Dulay, N.: GOANNA: State machine monitors for sensor systems (2008), `www.doc.ic.ac.uk/~lmostard/goanna`
16. European Commision 6th Framework Program - 2nd Call Galileo Joint Undertaking. Cultural Heritage Space Identification System (CUSPIS) (2007), `http://www.cuspis-project.info`
17. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming Wireless Sensor Networks with the TeenyLIME Middleware. In: Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007), Newport Beach, CA, USA, November 26–30 (2007)
18. Souto, E., Guimares, S., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: A publish/subscribe middleware for sensor networks. Personal Ubiquitous Comput. 10(1) (2005)
19. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to support sensor network applications. IEEE Network 18(1) (2004)