Towards Supporting Interactions between Self-Managed Cells

Alberto Schaeffer-Filho[1], Emil Lupu[1], Naranker Dulay[1], Sye Loong Keoh[1], Kevin Twidle[1], Morris Sloman[1], Steven Heeps[2], Stephen Strowes[2], Joe Sventek[2]

[1]Department of Computing, Imperial College London
{aschaeff, e.c.lupu, n.dulay, slk, kpt, m.sloman}@doc.ic.ac.uk

[2]Department of Computing Science, University of Glasgow
{heeps, sds, joe}@dcs.gla.ac.uk

Abstract

Management in pervasive systems cannot rely on human intervention or centralised decision-making functions. It must be devolved, based on local decision-making and feedback control-loops embedded in autonomous components. We have previously proposed the self-managed cell (SMC) as an architectural pattern for building ubiquitous applications, where a SMC consists of hardware and software components that form an autonomous administrative domain. SMCs may be realised at different scales, from body-area networks for health monitoring, to an entire room or larger distributed settings. However, to scale to larger systems, SMCs must collaborate with each other, and federate or compose in larger SMC structures. This paper discusses requirements for interactions between SMCs and proposes key abstractions and protocols for realising peer-to-peer and composition interactions. These enable SMCs to exchange data, react to external events and exchange policies that govern their collaboration. Dynamically customisable interfaces are used for encapsulation and interaction mediation. Although the examples used here are based on healthcare scenarios, the principles and abstractions described in the paper are more generally applicable.

1 Introduction

Management in pervasive systems cannot rely on human intervention or centralised decision-making functions. The former because pervasive devices must be usable by non technically savvy users. The latter because pervasive devices are mobile and cannot refer to centralised management applications for re-configuration and adaptation directives. Systems such as body-area networks of sensors and actuators for monitoring a patient's health, unmanned vehicles or fleets of vehicles must be autonomous and continuously adapt to changes in their environment or in their usage requirements. They must therefore be *self-managing* with local decision and feedback control to enable seamless adaptation. Whilst this structuring in autonomous entities is a necessity in pervasive environments, it has also been advocated as a means of constructing large distributed systems and networks. In essence, this is the proposition of autonomic computing [10]. To an extent, this proposition goes against the network management tradition, which focuses on the functional integration of management components across an entire corporate system and relies on centralised network operations centres manned by human administrators.

We have previously introduced the concept of a *Self-Managed Cell (SMC)* as an architectural pattern for building ubiquitous computing applications [11]. A SMC consists of a set of hardware and software components which form an autonomous administrative domain. SMCs implement a policy-driven feedback control-loop that determines which management and re-configuration actions should be performed in response to events of interest such as device failures, context changes or changes of state in the SMC's resources. The policy approach is itself based on previous work on policy-based management at Imperial College [5, 16]. SMC examples include body-area networks for health monitoring, unmanned vehicles, or control of pervasive spaces such as rooms, buildings or urban environments.

Although self-managed cells are autonomous, they must be able to interact with each other in complex ways, federate or compose into larger structures. For example, a body-area network monitoring a patient's health may comprise "smart" sensors and complex diagnosis devices that are SMCs in their own right. In the same way, SMCs controlling a smart-room will be aggregated under the control of a house SMC and autonomous unmanned vehicles may

be aggregated into fleets with a common mission. A body-area network SMC may interact with a number of other *peer* SMCs such as the SMC running on the PDA of a nurse, a doctor or other health-care worker, or the SMC controlling the room in which the wearer is present.

We propose here a way of realising "cross-SMC" interactions that enable complex collaborations between SMCs in either peer-to-peer or compositional settings. This permits realising scalable pervasive environments in which SMCs can aggregate into larger structures and engage in ad-hoc peer-to-peer collaborations. We focus on the basic abstractions for interactions in terms of exchanges of data, events and policies between the SMCs and discuss the main design decisions and architectural choices. Goal-driven collaborations relying on distributed planning approaches remain part of our plans for future work.

The examples used here are derived from requirements for e-Health. However, the principles and results are more generally applicable to other pervasive environments. In particular, we are currently applying them within the context of self-management for fleets of unmanned vehicles as well as management of large virtual organisations.

This paper is structured as follows: Section 2 discusses related work. Section 3 describes the SMC architectural pattern while Section 4 presents the SMC interactions framework. Section 5 details the behavioural specification of cross-SMC interactions. Our prototype and early results are described in Section 6. Concluding remarks and future work are presented in Section 7.

2 Related work

Although several studies have been devoted to designing frameworks for pervasive spaces, they tend to share two limitations: they focus on pervasive spaces of a relatively fixed size (e.g., a room) and they fail to cater for dynamic interactions between pervasive spaces. Both issues are however key points in the SMC design.

Often, research studies assume pervasive spaces of a relatively *fixed size*. For example, Gaia [13] seeks to extend the traditional operating system concept, by providing a view of a meta-operating system but focuses on room-sized environments. On the other hand, ISAM [1] aims to address resource management and application adaptation and focuses on large-scale multi-institutional environments. One.world [7], in turn, provides a less sophisticated infrastructure that enables applications to adapt to context changes but focuses on small room-like environments. iROS [8] emphasises the ability to integrate *"legacy"* applications but relies on centralised servers and limits itself to room-sized environments. Oxygen [12] is mainly concerned with how users interact with the system. It mentions the dynamic establishment of collaborative regions, which can be room-sized or even campus-wide but does not detail how this is achieved.

The second aspect relates to the federation of pervasive spaces. Whilst much of the literature focuses on the architecture of pervasive spaces and their supporting services, less attention is paid to the *interactions and collaboration* between such spaces. Gaia recognises the importance of federating Gaia spaces, but this is not part of the core view of its *active spaces* and is regarded as future research. ISAM assumes that the neighbourhood of each *execution cell* is configured by an administrator, and remains static for most of the time. iROS deliberately assumes a single pervasive space. Finally, Oxygen recognises the importance of establishing collaborative regions. However, details on how they are established are scarce. Furthermore, it is not clear whether this can be extended to collaborations between collaborative regions, where collaborative regions interact with each other.

In contrast, we consider the SMC as an architectural pattern applicable at different levels of scale, ranging from body-area networks, to large-scale virtual organisations. SMCs are expected to dynamically discover and collaborate with other SMCs, whilst most other projects focus on a single-size, single-instance perspective.

The IBM autonomic manager has some similarity to our SMC approach in that it autonomously manages a set of resource, while exposing a management interface to other autonomic managers, as though it is a managed resource. However, interactions between SMCs are considerably more sophisticated than the simple resource sensor/effector interface described in [3].

3 Self-managed cell architectural pattern

To provide autonomous management in pervasive environments, we have introduced the self-managed cell (SMC) as the basic building block of our pervasive systems [11]. A SMC consists of hardware and software components which form an administrative domain that is able to operate autonomously. Components (also referred to as managed resources), include physical sensors and actuators, devices such as PDAs, Gumstix, mobile phones and computers as well as software services and components within those devices. They are heterogeneous in nature and must therefore be accessed by the SMC's management services through *adapter* objects that provide a uniform management interface and hide the specifics of the interactions with those components. A typical set-up we use for health-care monitoring comprises a Gumstix[1] device hosting management services that controls several sensors (e.g., heart-rate, temperature, acceleration) hosted on BSNs (Body Sensor

[1] http://www.gumstix.com

Nodes) [2] as well as other devices such as diagnostic devices hosted on PDAs or other Gumstix. Communication with BSN nodes typically occurs through IEEE 802.15.4 radio links while communication between Gumstix devices or with PDAs occurs through Bluetooth or Wi-Fi.

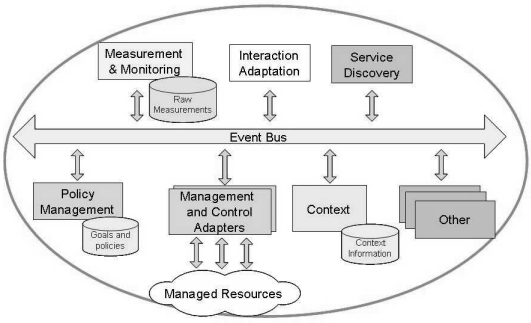

Figure 1. Self-managed cell architecture

A SMC comprises a dynamic set of management services that are integrated through a common publish/subscribe *event bus* (Figure 1). This has the advantage of de-coupling the services, as an event publisher does not require prior knowledge of the recipients when sending a message, and permits adding new services to the SMC without disrupting the behaviour of existing ones. The *core services* of a SMC are: the *event service*, the *policy service* and the *discovery service*; however, other services such as context, authentication, accounting or application specific services may be used in different SMCs. The SMC's core services implement a policy-driven feedback control-loop (Figure 2) [15] in which changes of state in the managed resources or changes of context are published on the event bus and trigger the execution of *obligation* policies in the form of *event-condition-action* rules that determine which adaptation actions need to be performed in response to the events.

The *policy service* caters for two types of policies: *obligation policies* that define the management actions that must be performed in response to events, and *authorisation policies* that specify which actions are permitted on which resources and services. Policies can be added, removed, enabled and disabled to change the behaviour of a SMC without interrupting its functioning. The managed objects to which policies apply can be internal SMC resources, adapters for external services or policies themselves. All managed objects are kept in a domain structure that implements a hierarchical namespace similar to a file system; however, domains may overlap and a managed object may belong to several domains. The SMC policy service is based on the Ponder2[3] system, which in addition to policies can interpret nested sequences of commands that identify the managed object to be used and parameters or sub-elements within the XML that are to be sent to the object.

[2] http://vip.doc.ic.ac.uk/bsn/

[3] http://www.ponder2.net

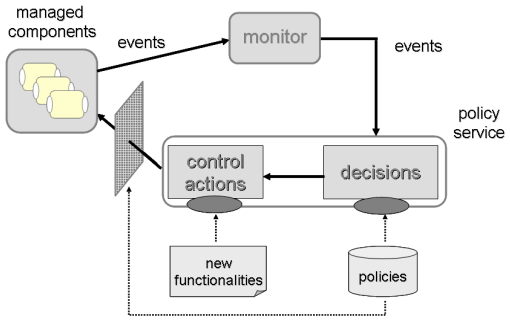

Figure 2. The feedback control-loop

The *event bus* disseminates the events needed to trigger obligation policies. Events generated by managed resources and by the services are transmitted to all the subscribed services enabling them to react concurrently to event occurrences within the SMC. Asynchronous events are well suited to pervasive systems in which most of the applications are event driven. However, we do not require that all interactions between SMC services be event driven.

The *discovery service* is used to detect new devices which are capable of joining the SMC, e.g., sensors and other SMCs in the vicinity. It interrogates new devices to establish a profile describing the services they offer and generates an event describing the addition of a new device for other SMC components to use it as appropriate. The discovery service is also responsible for vetting new devices before accepting them in the SMC and managing the SMC's membership, as it is necessary to distinguish transient failures which are common in wireless communications from permanent departure from the SMC (e.g., device out of range, switched off, or failure). When a new device is discovered, policies are used to decide in which domains the device and its accessible components should be placed. Policies applying to those domains will then automatically apply to the respective components.

4 SMC interactions

Although SMCs are autonomous, they need to interact with each other and aggregate into complex structures to scale to larger systems. Such interactions must be established autonomously with little or no user intervention.

4.1 Interaction requirements

A SMC managing a patient's health needs to interact with numerous *peer* SMCs. During home visits by a nurse

or other health-care practitioner it needs to permit the nurse SMC to access the data acquired on the patient's current physiological condition and must notify the nurse of events occurring within the patient SMC. The patient SMC may also need to be notified of events occurring within the nurse SMC, for example the fact that the nurse has started a specific diagnostic procedure. The nurse may need to load policies for execution by the patient e.g., for defining new thresholds or alert behaviour. Similarly, the patient SMC may need to load policies onto the nurse SMC e.g., to trigger re-calibration of the patient sensors if needed. Similar requirements would occur when the patient SMC encounters a SMC controlling devices in a General Practitioner's (GP) clinic, however, a GP may have additional access to the patient's resources e.g., the ability to change dosage on drug delivery pumps. Other *peer-to-peer* interactions may occur between the patient SMC and SMCs surrounding it such as environmental monitoring (e.g. pollen count, allergies), pharmacies, or other public services.

We can draw several requirements from the example above. Firstly, SMCs must detect the presence of peer SMCs and decide autonomously whether to establish an interaction. Interactions between peer SMCs require a SMC to be able to invoke operations on its peer, to receive event notifications from its peer as well as to notify its peer of selected internal events occurring within itself. More complex interactions may require exchanges of policies between the SMCs, if a SMC can request another to behave in a specific way. Secondly, the interface exposed to a peer SMC may include only a subset of the available operations and events depending on the kind of SMC and the role (e.g. doctor or nurse) it can play in the interaction. Finally, a SMC may wish to expose the resources it possesses though an external interface and may choose to mediate the interactions with those resources.

Whilst peer-to-peer interactions occur frequently as SMCs interact with neighbouring autonomous components, *composition* interactions enable grouping SMCs into larger autonomous structures and scaling SMC management to larger environments. Composition encapsulates a SMC, with its own resources, as a managed resource within the containing SMC. This implies that the SMC can be programmed by the containing SMC in terms of policies that it must enforce. Moreover, the device may expose to its containing SMC a management interface for re-configuration. For example a diagnostic device may be part of a body-area network that will load new decision algorithms and new policies into it. Similarly, larger sensors may be autonomous components and thus SMCs in their own right. For example, even BSN nodes can be connected to multiple analogue sensors and support internal event-based interactions. We have also implemented a basic policy service for BSNs to cater for adaptation to changing circumstances [9].

Composition also implies that the contained SMC behaves as a managed resource within the outer SMC and ceases to advertise itself independently. Interactions between the contained SMC and external SMCs are subject to the authorisation and possibly mediation from the outer SMC which may require preventing access to them from the outside environment. A SMC cannot be contained, i.e. treated as a managed resource, by more than one containing SMC, although it may interact with other SMCs for application purposes subject to authorisation from its managing SMC. Although a contained SMC is a managed resource, it must retain control of the interfaces it exposes and the policies it accepts from its managing SMC. This is for reasons of integrity rather than security as it is important to ensure that an autonomous device cannot be compromised i.e., devices preserve their autonomy.

Note that composition interactions have similar requirements to peer-to-peer interactions in terms of permitting invocations, raising and receiving events, and exchanging policies between the contained and the containing SMC. The differences between composition and peer-to-peer interactions lie in the degree of access permitted i.e., which methods and events are exposed and which policies are accepted from the containing SMC. The second important difference is that there can be only a single containing SMC and thus some of the methods and events are guaranteed to be invoked by a single entity. Finally, the third important difference is that a contained SMC ceases to advertise itself and thus its interactions with other devices are governed by its containing SMC. This allows the outer SMC to selectively hide the complexity of the composed structure, and only expose selected functionality in its external interactions (for example, the patient SMC would expose its sensors to the doctor, but hide them from other patients).

A SMC's interface may need to change dynamically as the SMC may acquire or lose functionality, for example through the addition/failure of a particular sensor or composition with a new SMC. To achieve this we are using a variation of the *role object pattern* [4] that allows to dynamically add new functionality to a *core* object by associating with it a new capability.

4.2 Customised interfaces

As discussed above, a SMC must determine which functions it wishes to export to its peers, and do so through specific *customised interfaces*. The customised interface exposed depends on the the kind of SMC it is interacting with (e.g. doctor, nurse) and even that SMC's identity. Although it would be possible to expose all the functions on a single interface and use authorisation policies to restrict access from external entities, this is insufficient for two reasons: first, the set of functions that a SMC *can* expose depends

on the resources it possesses (which may vary), and second, exposing an interface externally provides information about the function of the SMC. In medical scenarios this has privacy implications for the patient concerned. Therefore, an external SMC should *see* only those functions that a SMC wants to expose in a *customised interface* generated specifically for that interaction. Note that this interface could be generated from the SMC's authorisation policies.

A SMC typically *mediates* interactions between its resources and other SMCs. This mediation is implemented by a proxy object which maps the functions exposed in the customised interface to internal operations on the SMC's resources (e.g., a method named *"readTemperature"* is mapped to the *"/sensor/temp.read"* operation on an internal sensor). This approach avoids exposing the internal resources directly and permits realising more complex transformations e.g., of the parameters received or the result returned, when interacting with other SMCs. Additionally, this approach permits controlling which methods are exposed by adding or removing the respective mappings. In our prototype the interface exposed by a SMC is implemented as a managed object that provides the functionality of the proxy object described above. For example, the interface exported by a patient to a nurse will also act as a proxy for the invocations where the nurse tries to access the internal resources (e.g. sensors) of the patient.

In addition to invocations on a remote SMC, asynchronous communication through events is an important means of interaction between SMCs. Thus, an *interface* defines: *(a) events*, which can be published externally by the SMC (i.e., to which external SMCs can subscribe); *(b) notifications*, which are external events of which the SMC can be notified (i.e., that external entities publish within the SMC); and *(c) operations*, which are the methods that can be invoked on the SMC by a remote entity. Figure 3 shows the XML commands for generating an interface that a patient may export to a doctor SMC. It specifies that the doctor SMC may receive *monitoringReady* events and may raise *startMonitoring* and *stopMonitoring* events within the patient SMC. Additionally, the doctor SMC can invoke the *readECG* and *scheduleTask* operations on the patient SMC.

```
<create>
  <event name="monitoringReady"
           localEvent="ready"/>
  <notification name="startMonitoring"
           localEvent="start"/>
  <notification name="stopMonitoring"
           localEvent="stop"/>
  <operation name="readECG"
           localOp="/local/hearBeatSensor.read"/>
  <operation name="scheduleTask"
           localOp="/local/javaTimer.createTask"/>
</create>
```

Figure 3. Customised interface of a patient

Figure 3 also shows the mappings of the events, notifications and operations to local resources within the patient SMC. These mappings would not be visible to the client SMC when it queries the specification of the interface. Note that new operations, events and notifications can be added or removed from the interface definition, to change the functions exported.

4.3 Interaction establishment

Interaction establishment is initiated as a result of a *newSMC* event being generated by the SMC's discovery service [11], as depicted in Figure 4. This event contains the name of the discovered SMC, its profile, which identifies its type (e.g. patient, doctor or sensor), and a generic interface. The *generic interface* is application independent and common to all discoverable SMCs. It defines the operations necessary for exchanging customised interfaces and establishing the interaction.

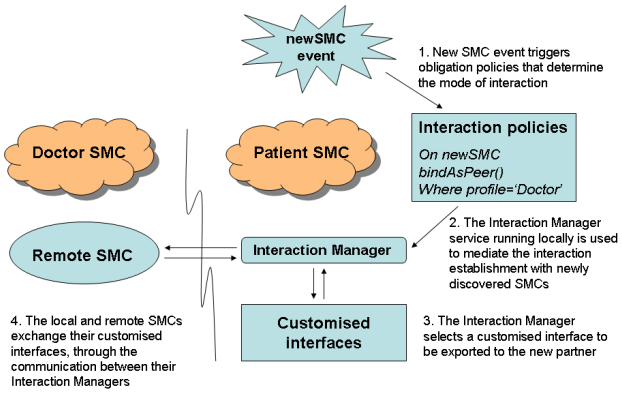

Figure 4. Interaction establishment overview

Obligation policies triggered by the *newSMC* event determine whether to establish a peer-to-peer or composition interaction. The *interaction manager* is a service running locally in each SMC, and it is used to bootstrap interactions. For example, the policy shown in Figure 5, deployed in a *patient* SMC, specifies that a peer-to-peer interaction should be established when discovering a *doctor* SMC. It achieves this by invoking the method *bindAsPeer* in the local interaction manager. Similarly, a composition relationship can be established with a *sensor* through the *bindAsResource* method.

The interaction managers in the two SMCs then exchange their *customised interfaces*. The interaction manager from the discoverer SMC selects a customised interface based on the remote SMC's profile and sends it to the interaction manager in the remote SMC. Similarly, the remote SMC selects a customised interface and returns it to the discoverer if it accepts the interaction. Each SMC stores

```
<create type="obligation" event="/event/newSMC">
  <arg name="name"/>
  <arg name="genericInterface"/>
  <arg name="profile"/>
  <condition>
    <eq>!profile;<!--  -->doctor</eq>
  </condition>
  <action>
    <use name ="/SMCCore/interactionManager">
      <bindAsPeer name="!name;"
                  genericInterface="!genericInterface;"
                  profile="!profile;"/>
    </use>
  </action>
</create>
```

Figure 5. Interaction policy

the received interface in its local domain structure. Note that simple BSN nodes will not initiate discovery but will only respond to being discovered.

4.4 Domain structure and roles

When two SMCs interact, obligation policies specified in one SMC may define invocations to be performed on the other SMC. Furthermore, as discussed in Section 4.1, SMCs may exchange policies between them. One SMC may request that the remote SMC behave in a particular way and can achieve this by sending to the remote SMC a set of obligation policies. In both cases policies are written in terms of the events and actions of a remote SMC before that SMC comes into proximity. Therefore, a SMC must know the interface it expects an encountered SMC to have e.g., a patient will have a description of the interfaces it expects a doctor or a temperature sensor to have. Policies applicable to that SMC can then be specified in terms of this *expected interface*.

We use *roles* as placeholders within the local domain structure for SMCs discovered at run-time. Roles are associated with an *expected interface*, that defines the operations, events and notifications that remote SMCs are expected to provide in order to be assigned to that role. Then, policies for that role can be defined in terms of the expected interface's events and operations as explained in the following section. When an interaction is established, the remote SMC is assigned to a role based on its profile. At this point the SMC verifies that the customised interface offered by the remote SMC provides a superset of the elements required in the expected interface. Policies already specified for that role will then apply to the remote SMC. Several SMCs might be assigned to the same role (e.g., a doctor can interact with several patients). Additionally, the role is also a suitable place to store the customised interface that can be exported to that SMC. For example, the doctor will store in the *patient* role the interface it expects patients to provide and the interface that it will export towards patients. Figure 6 illustrates a domain structure for a patient SMC with roles for interacting with *sensors*, *doctors* and *nurses*.

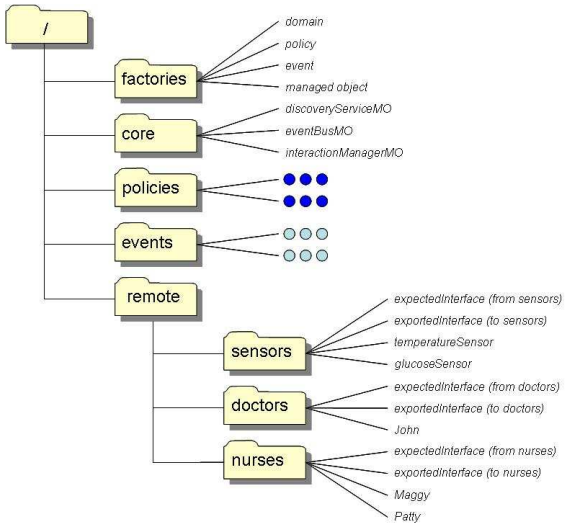

Figure 6. Domain structure and roles

In summary, a role provides three functions to facilitate cross-SMC interactions: *(a)* it identifies the *expected interface* that remote SMCs must provide to be assigned to that role; *(b)* it identifies the interface that the SMC will export to remote SMCs of that kind (e.g., the *setTemperatureThreshold* method is to be exported to a doctor but not to a nurse); and *(c)* it defines a placeholder for the remote managed objects and SMCs for which policies are specified.

5 Missions: behavioural specifications of SMC interactions

After the initial steps for establishing an interaction have been performed, the SMCs can start their collaboration (collaborations between ubiquitous entities is also called *recombinant computing* [6]). This permits a SMC to be notified of and react to events occurring in the other SMC and thus extend its control-loop.

Complex interactions can be defined by exchanging policies between SMCs. These policies define how the SMCs should behave in the context of the interaction in terms of sending notifications to other SMCs, and reacting to both internal events and external notifications by invoking management actions locally or on remote SMCs. We introduce the concept of a *mission* as a means of grouping the duties of the remote SMC in the interaction specified in terms of the obligation policies it must enforce. Thus, a mission is a group of obligation policies specified in terms of the interfaces of two or more interacting SMCs.

A mission is normally specified before the target SMC has been discovered and is therefore defined in terms of the

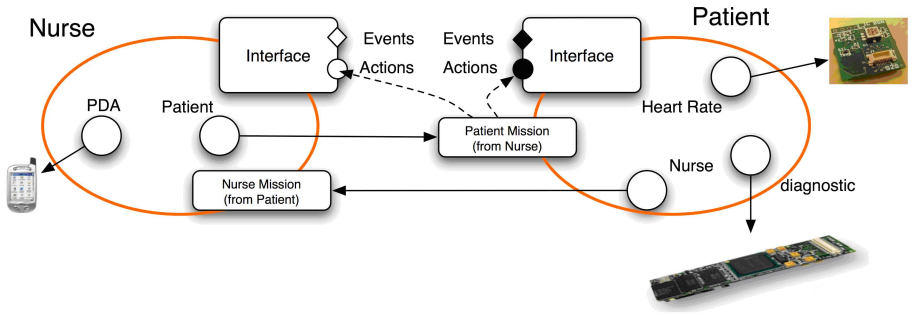

Figure 7. Missions across SMCs

expected interfaces of the SMCs involved. As described in Section 4.4 these interfaces are associated with roles within the current SMC so we informally say that a mission is written as a function of the roles of the SMCs involved. The obligation policies i.e., event-condition-action rules grouped within the mission are thus written in terms of the events and actions defined in the expected interfaces associated with the roles. When a new SMC is discovered and assigned to a role, obligation policies within the current SMC determine which missions should be instantiated on it.

As shown in Figure 7, when the nurse SMC discovers a patient's body-area network SMC, it instantiates a mission on it if permitted by the latter. Similarly, the patient may instantiate at the nurse a mission defining the policies it expects the nurse to fulfil in the interaction. Note that it is not necessary that both missions be initiated.

In essence, missions are a constrained form of programming a remote SMC and are akin to a form of dynamically loaded code. Thus, before executing this code i.e., instantiating the mission and its policies, the receiving SMC must validate the received mission through a procedure detailed in Section 5.2. This is necessary in order to avoid that the received mission compromises the integrity of the SMC either accidentally or maliciously.

5.1 Specifying a mission

Figure 8 shows an example of a mission between a nurse and a patient for *ECG monitoring*. This specification is a template taking as arguments the roles (and implicitly their interfaces) to which it relates as well as two additional arguments (*time* and *frequency*) that will be specified at run-time when the mission is instantiated. Conceptually, the mission specifies the obligations that patients must enforce in order to enable a nurse to perform an ECG. The argument values refer to the specific nurse and patient that are to interact and are given upon instantiation.

This mission comprises an obligation policy triggered by a *startMonitoring* event received from the nurse, which requests the patient to schedule two tasks: one that reads the patient's ECG for a specified time and at a specific frequency, and the other that notifies the nurse when the monitoring has finished. Thus, this mission relies on the methods *scheduleTask* and *readECG* that are expected to be present in the patient's interface, and on events that must be either generated or received by the SMC. The method *notify* used in this example is a generic method through which an entity can publish an event to a SMC. Its argument must be one of the notifications defined in the receiving SMC's interface. The method *load*, not shown in the example, is used to load a mission and is also generic to all SMCs.

```
<create>
  <arg name="nurse" type="interface/nurse"/>
  <arg name="patient" type="interface/patient"/>
  <arg name="time" type="integer"/>
  <arg name="freq" type="integer"/>
  <policy name="ECGMonit" event="!nurse;.startMonitoring">
    <action>
      <use name="!patient;">
        <scheduleTask freq="!freq;" time="!time;">
          <use name="!patient;">
            <readECG />
          </use>
        </scheduleTask>
        <scheduleTask delay="!time;">
          <use name="!nurse;">
            <notify event="!patient;.monitoringReady"/>
          </use>
        </scheduleTask>
      </use>
    </action>
  </policy>
</create>
```

Figure 8. Patient monitoring mission

When a mission is specified, a first verification of the mission is performed, to ensure that it complies with the *expected interfaces* of the roles involved. This is achieved by parsing the different policies and building a dependency table that includes all the events and actions which policies refer to as well as the role interface to which they are expected to belong. This table is then checked against the ex-

pected interfaces associated with those roles. The mission is considered valid if all the dependencies are satisfied.

5.2 Loading a mission

When a mission is instantiated at a remote SMC, that SMC receives the mission specification and the parameters and must instantiate the obligation policies contained in the mission within its own scope. However, before doing this it must check that the mission is well-formed, that all the mission's dependencies can be satisfied within its local environment and that the policies received do not conflict either with its own policies or with policies originated from other missions. We discuss here how the first two requirements are addressed. Although we have developed several algorithms for policy conflict analysis [2] their implementation has not yet been integrated in the SMC framework.

5.2.1 Step 1: Check that the mission is well-formed

A source SMC may maliciously or accidentally embed additional code in the mission and attempt to load it in the target SMC. Therefore, the first step in validating the mission is to check that it is well-formed; namely, that it contains only arguments and obligation policies and that it is syntactically correct. This includes inspecting the policies in the mission and verifying that they use solely operations and events pertaining to the role interfaces given as arguments. A policy attempting to invoke operations on other objects will generate an error and abort the instantiation of the mission. This ensures that the mission is self-contained and prevents malicious SMCs from "guessing" operations available in the target SMC.

5.2.2 Step 2: Check mission parameters

When a mission is instantiated by one SMC on another e.g., the doctor instantiates a mission on a patient, the argument values sent to the patient are the generic interfaces of the SMCs involved. This is because in missions involving more than two SMCs each SMC has a different view of the capabilities of the SMCs with which it interacts. For example, in Figure 9, where the dashed lines represent interface exchanges and the solid lines represent mission instantiation, the doctor's interface to a nurse SMC (*NS*) may be different from the interface that the patient has to the same nurse SMC (*NS'*). Thus, if the doctor wants to instantiate in the patient SMC a mission that involves interactions with the nurse SMC it will pass as argument to the mission the generic interface of the nurse SMC. It will be up to the patient SMC to contact the nurse SMC and obtain an interface to that SMC if it does not have one already.

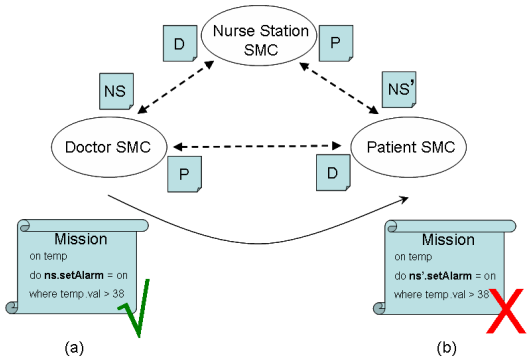

Figure 9. Mission requirements satisfied by the interfaces known to the doctor (a) but not by those known to the patient (b)

Thus, when receiving a mission, the recipient SMC must establish the interactions and obtain a customised interface from all the SMCs given as mission parameters.

5.2.3 Step 3: Check mission dependencies

A SMC that has received a mission must check the policy dependencies (i.e., events, notifications and operations used in the mission's policies) against the interfaces that it has for the SMCs given as mission parameters. This can be achieved by computing the policy dependencies and checking them against either: *(a)* expected interfaces it knows for those SMCs or *(b)* the customised interfaces it has obtained from the remote SMCs once it has established an interaction with them. The former is sufficient because an interaction will subsequently be established with a SMC only if the customised interface received from that SMC is more specific (i.e., supports a superset of events, notifications and operations) than the expected interface for that SMC. This approach enables delaying binding to the remote SMC until the binding is actually required, but is more restrictive since it requires the SMC to have an expected interface for the remote SMC and the customised interface may offer additional operations that are not present in the expected interface. The second approach is more permissive as it allows the mission to contain policies that use operations not present in the expected interface but requires establishing an interaction with the remote SMCs when the mission is received. In terms of our previous example (Figure 9) the doctor instantiates a mission on the patient which requires the patient to interact with a nurse. The patient will verify this mission against the expected interface it has for nurses or may obtain a customised interface from the nurse given as parameter and check the policy dependencies of the mission against this interface. In our implementation, we have chosen the latter option as it enables the doctor to deploy

a mission to the patient that takes advantage of operations provided by the specific nurse the patient will be interacting with (e.g., the nurse in that specific GP clinic or ward).

5.2.4 Step 4: Instantiate mission

If all the above steps succeed, the receiving SMC instantiates the obligation policies contained in the mission, using the argument values provided by the source SMC. Arguments that are basic types require a trivial variable substitution, and the interfaces are substituted with the appropriate adapter object for the remote SMC. These policies can now trigger internal actions within the patient based on events occurring in either the doctor or the nurse SMC or trigger remote invocations on the doctor and nurse SMCs in response to events occurring within the patient SMC. It is thus possible to encode complex collaborations between the SMCs and both the patient and the nurse SMCs can deploy missions to each other in a similar fashion.

For a SMC to deploy a mission to another it must be authorised to do so by authorisation policies in the recipient SMC. Furthermore, the actions specified in the mission's policies also need appropriate authorisations in the receiving SMC. The access control framework for the SMC is presented in detail in [14].

6 Implementation and discussion

Our current implementation caters for most of the concepts described in this paper, including the implementation of the obligation policies in the Ponder2 interpreter, the selection and exchange of interfaces between interaction managers, the deployment of missions, the verification of missions by the recipient SMC, and their instantiation. The current implementation of the interaction framework is in Java, however porting it to the Gumstix remains to be done. Integration with the lightweight implementation of the SMC core services [9] for constrained BSN sensors will be addressed in the future.

When instantiated, the SMC starts the policy service, which in turn creates a local instance of the interaction manager and of the other core services. All managed objects, including missions, interfaces and roles are created through factory objects that can be loaded on-demand and can be invoked at run-time using XML based commands to create new instances. This also enables us to cater for scenarios in which interactions occur spontaneously and the role descriptions need to be loaded dynamically.

In order to build the scenario described in this paper, we have pre-deployed in the SMCs: (a) the policies that determine which type of interactions need to be created (Figure 5), (b) the role definitions for doctors, patients and sensors, and (c) the mission specification illustrated in Figure 8. Customised interfaces are at present specified rather than generated from authorisation policies as the access control framework has only recently been implemented.

The discovery service discovers new devices over IEEE 802.15.4, Wi-Fi or Bluetooth. If the new device is a SMC (as indicated by its profile) a *newSMC* event is generated that will trigger the policies that decide if an interaction should be established, and whether it is a peer-to-peer or a composition one. The interaction managers in the two SMCs then exchange customised interfaces, assign the interface received from the remote SMC to the appropriate role and generate a local *newSMCBinded* event at each SMC. This triggers policies that determine which missions should be loaded on the partner SMC. The verification procedure described in Section 5.2 is then carried out before the policies included in the mission are instantiated.

We have tested our implementation in a small-scale set-up consisting of three SMCs: *ECG sensor*, *patient* and *doctor*. The ECG sensor and the patient establish a composition interaction while the patient and the doctor establish a peer-to-peer interaction. The doctor is then given access to the ECG sensor but the access is mediated by the patient SMC. Missions have been loaded from the patient SMC to the ECG sensor, and from the doctor to the patient SMC. We were able to test the functionality of our implementation running on workstations, but not on Gumstix yet. Therefore, we are not presenting here performance results, as the measurements obtained in workstations would not reflect a realistic situation. Although the use of XML may add some overhead, we have chosen to use it in order to improve interoperability with client applications that can issue commands to any managed objects within the scope of the policy interpreter by generating the appropriate XML. Overall, our evaluation indicated that the SMC framework is suitable for realising self-management but may not be suitable for real-time application data such as streaming measurements from the sensors.

For body-area networks for health monitoring we have implemented our own discovery service and event bus. Although a number of applications offering similar functionality exist, we needed implementations that could scale down to small devices and could be used in conjunction with policy decisions. However, when using the SMC pattern in larger scale environments, more efficient implementations (e.g., such as SIENA for the event bus) can be used.

7 Concluding remarks and future work

Using the SMC as an architectural pattern, basic SMCs can be dynamically assembled into larger and more complex structures. This allows SMCs to scale-down to individual devices, and scale-up to cater for larger pervasive applications. Policies provide not only a means of adapt-

ing to events corresponding to changes of state in the managed resources (e.g. failures or performance degradation) but also to define how SMCs should behave when they encounter new autonomous entities and in particular new SMCs. Through the concept of *missions*, obligation policies in the form of event-condition-action rules can be used to enact a rich spectrum of collaborations between SMCs.

Although we started by considering peer-to-peer relations as intrinsically different from composition relations, their requirements proved to be quite similar in terms of the need to cater for cross-SMC invocations and exchanges of events and policies. What varies substantially are the invocations that are permitted, events that can be exchanged and policies that can be accepted. The other substantial difference lies in the ability of a composite SMC to hide its underlying complexity. The use of customised interfaces and mediated interactions have proved useful in terms of both reducing complexity and handling interactions with heterogeneous components. Mediated interactions are essentially for management activities and do not preclude establishing direct interactions with resources for application purposes.

The SMC pattern, which is based on a dynamic set of services integrated around a publish/subscribe event bus and uses policy-based adaptation, has proved applicable to a wide spectrum of environments. Although this paper has focused on health monitoring, we have worked on similar applications for fleets of unmanned autonomous vehicles (the term *mission* originates from there) and in large distributed settings such as virtual organisations.

Finally, the abstractions and protocols defined here provide a basis on which SMC collaborations can be defined. However, further work remains to be done towards: *(a)* integrating in the current framework procedures for checking policy analysis, conflict detection and resolution, *(b)* integrating the current implementation with the access control framework and *(c)* extending the mission concept to both cascaded configurations and higher-level abstractions for SMC interactions. In respect to the latter we are currently working on the appropriate infrastructure for defining missions that can encapsulate other mission specifications e.g., the doctor would deploy a mission to the nurse SMC that itself contains the instructions for the nurse to load specific (sub-)missions to any of the patient SMCs that it encounters. Higher-level abstractions may include collaboration definitions whose enforcement is itself distributed and collaborations based on exchanges of high-level goals which require planning procedures and algorithms to be scaled down to embedded devices.

Acknowledgments

The authors wish to thank the UK Engineering and Physical Sciences Research Council for their support of this research through grants GR/S68040/01, GR/S68033/01 and EP/C547586/1.

References

[1] I. Augustin et al. ISAM - a software architecture for adaptive and distributed mobile applications. In *7th IEEE Symp. on Computers and Communications*, pages 333–338, Taormina, Italy, July 2002. IEEE-CS.

[2] A. Bandara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy sepcification and analysis. In *IEEE 4th Int. Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, Como, Italy, June 2003. IEEE.

[3] D. F. Bantz et al. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003.

[4] D. Bäumer et al. The role object pattern. In *Proc. Int. Conf. on Pattern Languages of Programs (PLoP 1997)*, Washington University, 1997. Technical Report WUCS-97-34.

[5] N. Damianou et al. The Ponder policy specification language. In *Proc. IEEE Workshop on Policies for Distributed Systems and Networks*, pages 18–39, Bristol, U.K., Jan 2001. IEEE-CS.

[6] W. K. Edwards et al. Challenge: recombinant computing and the speakeasy approach. In *Proc. 8th ACM Int. Conf. on Mobile computing and Networking*, pages 279–286, New York, NY, USA, 2002. ACM Press.

[7] R. Grimm. One.world: experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, Jul.-Sep. 2004.

[8] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67 – 74, April-June 2002.

[9] S. L. Keoh et al. Policy-based management for body-sensor networks. In *Proc. 4th Int. Workshop on Wearable and Implantable Body Sensor Networks (BSN)*, LNCS, pages 92 – 98, Aachen, Germany, Mar. 2007. Springer.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan 2003.

[11] E. Lupu et al. AMUSE: autonomic management of ubiquitous systems for e-health. *J. Concurrency and Computation: Practice and Experience, John Wiley (to appear)*, 2007.

[12] Oxygen website, 2006. Available at: <http://www.oxygen.lcs.mit.edu/>.

[13] M. Roman et al. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, Oct.-Dec. 2002.

[14] G. Russello, C. Dong, and N. Dulay. Authorisation and conflict resolution for hierarchical domains. In *Proc. IEEE Int. Workshop on Policies for Distributed Systems and Networks*, Bologna, Italy, Jun. 2007. IEEE CS-Press.

[15] M. Sloman. *Network and Distributed Systems Management*, chapter Monitoring Distributed Systems, pages 303 – 347. Addison-Wesley, 1994.

[16] M. Sloman and E. Lupu. Security and management policy specification. *IEEE Network*, 16(2):10–19, Mar.-Apr. 2002.