Noname manuscript No.
(will be inserted by the editor)

Federating Policy-driven Autonomous Systems: Interaction Specification and Management Patterns

Alberto Schaeffer-Filho · Emil Lupu · Morris Sloman

Received: date / Accepted: date

Abstract Ubiquitous systems and applications involve interactions between multiple autonomous entities — for example, robots in a mobile ad-hoc network collaborating to achieve a goal, communications between teams of emergency workers involved in disaster relief operations or interactions between patients' and healthcare workers' mobile devices. We have previously proposed the *Self-Managed Cell (SMC)* as an architectural pattern for managing autonomous ubiquitous systems that comprise both hardware and software components and that implement policy-based adaptation strategies. We have also shown how basic management interactions between autonomous SMCs can be realised through exchanges of notifications and policies, to effectively *program* management and context-aware adaptations. We present here how autonomous SMCs can be composed and federated into complex structures through the *systematic composition of interaction patterns*. By composing simpler abstractions as building blocks of more complex interactions it is possible to leverage commonalities across the structural, control and communication views to manage a broad variety of composite autonomous systems including peer-to-peer collaborations, federations and aggregations with varying degrees of devolution of control. Although the approach is more broadly applicable, we focus on systems where declarative policies are used to specify adaptation and on context-aware ubiquitous systems that present some degree of autonomy in the physical world, such as body sensor networks and autonomous vehicles. Finally, we present a formalisation of our model that allows a rigorous verification

Neither the entire paper nor any part of its content has been published or has been accepted for publication elsewhere. It has not been submitted to any other journal.

Alberto Schaeffer-Filho
Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
E-mail: alberto@inf.ufrgs.br

Emil Lupu
Department of Computing, Imperial College London, United Kingdom
E-mail: e.c.lupu@imperial.ac.uk

Morris Sloman
Department of Computing, Imperial College London, United Kingdom
E-mail: m.sloman@imperial.ac.uk

of the properties satisfied by the SMC interactions before policies are deployed in physical devices.

Keywords Network management · Adaptive policy · Federation · Autonomic · Architectural pattern

1 Introduction

Ubiquitous systems typically comprise numerous devices, such as smartphones, robots and intelligent sensors and actuators that interact via wireless communications in applications such as body sensor networks for healthcare, teams of emergency workers and unmanned robots for disaster relief operations, sensor networks for environmental and infrastructure monitoring, etc. Such systems must operate continuously, but their configuration and management are too complex and cumbersome for non-technical users. Autonomic, self-managing techniques are therefore needed to build these systems and make them adaptive to context changes, failures and mobile components which join or leave the system frequently.

Previous work introduced the *Self-Managed Cell* (SMC) [1] as a paradigm for structuring the management of ubiquitous applications. An SMC consists of a set of autonomous hardware and software components, which may themselves be SMCs. In essence, an SMC provides an architectural pattern for the implementation of a MAPE-K loop [2] where management policies are used to specify adaptation strategies to be enacted in response to changes in context, failures, components joining or leaving the system, and authorisation policies restrict which resources and services can be accessed, including by other SMCs. We have applied SMCs in multiple areas including body sensor networks for e-health, autonomous vehicles, and management of larger networks and systems [3, 4]. For example, a typical SMC for health monitoring comprises a smartphone supporting user interaction and management services that control several intelligent *body sensor nodes*[1] (BSN) for monitoring heart rate, temperature and oxygen saturation, which may be SMCs themselves. The conditions monitored by the sensors may result in alerts to the user or actions on actuators such as a pacemaker or a drug delivery SMC. If the patient's state indicates a critical condition, a remote emergency/healthcare service could also be summoned. A healthcare worker may have a smartphone or netbook able to perform specific diagnostic services that require interaction with the patient's SMC or may update the patient's policies to modify the care strategy. Communication with BSN nodes typically occurs through IEEE 802.15.4 radio links while communication between smartphones occurs through Bluetooth or Wi-Fi. Similarly, disaster management situations require the collaboration of first-responder organisations such as police, ambulance and fire brigade but may also involve local authorities and charities such as the Red Cross. These have their own management structures and policies but still need to interact, coordinate activities, assign tasks and so on. Rescue operations may include unmanned vehicles to search dangerous environments. A team of collaborating unmanned autonomous vehicles (UAVs) could be used for search and rescue where each UAV can be seen as an SMC with basic capabilities for controlling movement and sensing the environment e.g., detecting chemicals, video camera, microphone for monitoring sound, GPS

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

for location sensing and infrared sensors for detecting physical obstacles. A UAV may support various communication options including Wi-Fi for interacting with other vehicles, satellite or cellular 3G (in urban environments) for long distance interactions.

In the above scenarios both peer-to-peer relations, e.g., between first-responder organisations, as well as compositions, e.g., in a healthcare body sensor network managed by the smartphone controller, need to be catered for. Some applications combine both peer-to-peer and composition relations such as a network of UAVs collaborating on a search and rescue mission, which may indicate a peer-to-peer relationship between them but also be composed into a single SMC with a controller which manages the mission, assigns tasks and controls access from/to the external world. Although the ways in which autonomous systems interact in such applications are endless, common patterns are often encountered in relation to:

- *Communication* and how the distributed entities exchange information;
- *Management* in terms of how entities are discovered, how tasks are allocated, how policies are exchanged;
- *Structuring* of the potentially large number of entities forming a complex SMC, e.g., composition, peer-to-peer or some form of federation and how interfaces of an outer SMC can be used to control access to inner SMCs.

These patterns can be identified, codified and used as building blocks in the creation of application-specific patterns as well as for the rapid assembly of SMCs into complex structures by instantiating appropriate patterns.

In this paper we describe how this can be achieved, how management patterns can be implemented and codified, how they can be formally checked for correctness and how interactions can be deployed. We advocate ways of structuring these collaborations, as well as the specification, instantiation and reuse of common *patterns* between SMCs. We show how the different SMC interaction patterns can all be expressed as combinations of three fundamental abstractions: *policy exchange*, *event forwarding*, and *interface visibility*. We further show how SMCs can dynamically exchange policies to prescribe how remote SMCs must behave; how SMCs can forward events which are required for communicating changes of context and triggering management policies in another SMC; and how SMCs are structured with respect to interface access, to enforce abstractions such as visibility and encapsulation. Patterns are used to prescribe the protocols through which the exchanges of policies, events and interfaces are achieved. The use of patterns supports the construction of SMC interactions in a methodical manner, by reusing simpler abstractions as design elements of a more complex interaction. To achieve this we have extended, integrated and consolidated work presented in earlier workshop and conference publications in a coherent framework [5–7]. In particular, we add to the SMC an extensible set of interaction patterns relating to the management structures, types of control and types of communication that are useful for federating autonomous SMCs; we individually discuss and give examples of the most common patterns in our catalogue; we present the formalisation of pattern-based SMC interactions; and we present a comprehensive evaluation, not only in terms of the performance of our implemented prototype but also in terms of the overall scalability of the model. Note that this paper deals specifically with the interaction specification and management patterns for composing SMCs. We do not cover all aspects of SMC integration regarding coordination, orchestration, governance and

decomposition. Although these are interesting research challenges in their own right, and some have been partially addressed in separate publications, further work is required to integrate them within the context of the work in this paper.

This paper is organised as follows: Section 2 presents the Self-Managed Cell and the basics of cross-SMC interactions. Section 3 introduces the use of patterns for systematically realising policy-based SMC interactions. Section 4 presents the formalisation of the model and the verification of SMC interactions. Section 5 describes our implementation and evaluation. Section 6 discusses the related work and Section 7 presents the concluding remarks.

2 Self-Managed Cells and their Interactions

2.1 Self-Managed Cell

The Self-Managed Cell (SMC) provides a paradigm for structuring the management of autonomous ubiquitous systems [1,3], and has evolved from previous work on policy-based management at Imperial College. An SMC facilitates easy addition or removal of components, caters for error prone sensors, failures and automatically adapts to the user's current activity or environment. It uses policies as the primary means of implementing adaptation but further decision making components and services (e.g., planning [8]) can be integrated as additional services. The SMC manages a set of heterogeneous components such as body sensor nodes, smartphones, Gumstix[2], robots or network elements and could range from body-area networks to large-scale distributed applications.

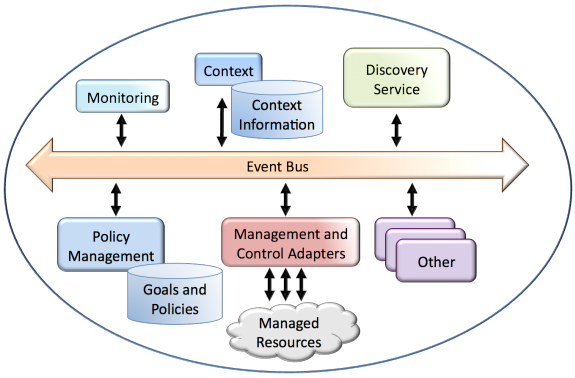


Fig. 1 Self-managed cell architecture comprising its core services

The SMC is itself a pattern that can be instantiated for different application areas using different implementations for its core services. For example, one implementation can be used for an individual device SMC such as a smartphone or an autonomous vehicle, and a different one when catering for the management of larger networked systems. The management functionality in the SMC is provided

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

through a dynamic set of *management services* (Fig. 1) integrated through an asynchronous *event bus* that carries management events between the components and services. A *policy service* that enforces adaptive and access control rules, and a *discovery service* that can discover new components and maintain the membership of the SMC are the core services that a minimal implementation requires. The event bus provides loose coupling between the services thereby realising an *extensible* architecture where additional services, e.g. for retrieving a specific type of contextual information, can be added as required [1]. Whilst a similar architecture may in some circumstances also be appropriate for realising the functional interactions between the SMCs components, we focus here solely on the *management* interactions. In particular, we do not consider the case where the event bus carries all the functional messaging traffic.

2.1.1 Discovery Service

The discovery service is used to detect new devices capable of joining an SMC, e.g. intelligent sensors or other SMCs, when they come into communication range. Typically, a sensor node may contain various resources such as storage, processing power, GPS receivers, cameras, etc, and discovery protocols are needed so other nodes can discover and use the desired resources [9]. In the SMC model, the discovered device is authenticated, and if necessary a secure channel is established for further communication. The device is interrogated to determine the capabilities it offers in terms of services or resources. This is achieved by exchanging SMC interfaces specifying the operations and events supported (see Section 2.2.2), described in a simplified notation based on Ponder2. The discovery service is responsible for maintaining the membership of the SMC, functioning as a *registry* – a reference to the discovered device is stored in a domain structure within the SMC in the form of a *managed object* (MO), which works as an adapter that abstracts the communication protocol, e.g. sockets, RMI, HTTP, between the discoverer and discovered SMCs. The discovery service broadcasts its *identity message* (id;type[;extra]) at frequency R. This enables the SMC to advertise itself to both devices and other SMCs, and enables current SMC members to determine whether they are still within reach of the SMC. Each member device unicasts its identity message at frequency D, and if the discovery service misses n_D successive messages from a particular device, it concludes that the device has left the SMC permanently. The discovery service thus detects when one of the SMC's resources has left and can distinguish between transient failures and permanent departures. The event describing the departure of one of the components is published on the event bus allowing all relevant management services to react to it concurrently.

2.1.2 Event Bus

A publish/subscribe event bus is used to communicate notifications, alerts and other events between managed resources and management services or between management services themselves. This de-couples services and resources, as event publishers do not need prior knowledge of the recipients when sending a message [10]. This also permits adding new services to the SMC without disrupting existing ones. The event bus must guarantee reliable event delivery since managed events are used to trigger adaptation and reconfiguration actions. For embedded

systems such as body-area networks we have developed a simple publish/subscribe event system supporting minimal functional requirements. In particular it supports at-most-once persistent event delivery (it attempts to deliver an event until it determines that the subscriber is no longer a member of the SMC) and content-based subscriptions, where subscriptions can be matched against any field of the event. Messages are routed by the event bus using *filters*, which match the subscriptions with the content of the events published. Furthermore, the event bus must guarantee that events from the same publisher will be delivered to the subscriber in the same order as they have been received by the router (FIFO ordering).

Our implementation of the event bus is aimed at resource constrained devices and thus does not intend to support the functionality of an Enterprise Service Bus (ESB) [11]. In larger scale systems where the event bus is not only used for management events but also for functional interactions between the managed resources and services an ESB may need to be used. Similarly, fine grained control over the information flow can be implemented in large scale publish/subscribe systems, for example as described in [12].

2.1.3 Policy Service

A policy service manages the policies specifying the behaviour of the SMC. Our implementation is based on the *Ponder2* framework[3], but we also provide a lightweight implementation that can run on BSNs and other constrained devices [13]. Ponder2 comprises a general-purpose object management system. It implements a policy execution framework that supports the enforcement of both obligation and authorisation policies. *Obligation policies* are event-condition-action rules which define the adaptive behaviour of an SMC — how discovered devices are controlled, how to recover from component failures, which adaptation strategies to apply when context changes, which events and notifications should be generated within the SMC or for the benefit of external SMCs such as collaborators in a mission. *Authorisation policies* specify the conditions for permitting other parties to access services or resources within the SMC. New policies may be downloaded into an SMC at run-time or existing policies may be enabled/disabled to change the adaptation strategy of an SMC. For example, the following policies could be specified for healthcare monitoring and recovery of a patient:

```
1.  on hr(level)
        if level > 100 do /os.setFreq(10min);

2.  auth+ /patient
        → /os.{setFreq, setMinVal, start, stop};
```

The first policy is an obligation which is triggered by a heart rate (hr) event produced by a body sensor. If the level measured is above a certain threshold, the policy reconfigures the monitoring frequency of the oxygen saturation (os) device. The second policy is an authorisation required to permit management of the oxygen saturation device. Policies are written in terms of *managed objects* (*MOs*), which are stored in a local domain that implements a hierarchical namespace within Ponder2. Ponder2 provides built-in support for the creation of a set of core managed objects, e.g. *events*, *policies*, etc, however the infrastructure is extensible

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

and allows the creation of user-defined custom managed objects, e.g. *adapters* for interfacing with a temperature sensor. Managed objects may also be held transparently in a remote Ponder2 system, and different underlying transport protocols are natively supported to facilitate remote communication, e.g. RMI, HTTP, etc. A command interpreter provided by Ponder2 supports a high-level configuration and control language called *PonderTalk*, which allows the invocation of actions on these managed objects.

2.1.4 Additional Services

Whilst the services described above constitute the minimal functionality of the SMC [1], the architecture described in Fig. 1 is extensible and other services may be added depending on the requirements of the application domain. A security service can implement authentication, secure device association and support confidentiality and anomaly detection [14]. A service to optimise performance and resource allocation could be added in more complex SMCs. Work on policy refinement [15], policy learning [16], orchestration/planning [17], QoS and conflict analysis [18] has been done at Imperial College for some time in the context of policy-based systems. These aspects could be integrated in the SMC, and the work presented here may need to be extended to cater for the additional interactions required for these services (e.g., propagation of constraints for orchestration and planning). However, we leave these aspects for future work and concentrate in this paper on the interactions required for the basic SMC functionality and the use of management patterns. A variety of communication and transport protocols may be necessary in order to interact with heterogeneous components in an SMC. For example, body sensors typically use IEEE 802.15.4 wireless links or bluetooth whereas smartphone controllers may include Wi-Fi and 3G for interactions with the infrastructure. To achieve this, adapter objects are created when the device is discovered and are used to provide a uniform interface to the SMC and convert interactions to the specific protocols supported by the devices.

2.2 Cross-SMC Interactions

Large systems need to be built in terms of SMCs that cooperate between them. This requires realising complex SMC structures including peer-to-peer collaborations, federations and compositions, which need to be defined in terms of the interactions between the SMCs. Such interactions depend on the type of the SMCs but also on the context in which the interactions take place. Thus, the implementation of the interactions cannot be hard-coded but must be extensible and configurable. Cross-SMC interactions rely on a set of basic underlying mechanisms including the interfaces SMCs expose to each other, the roles interacting SMCs play in each other's structure, and events and policies exchanged between them. Policies specify the obligations of an interacting party in the collaborations. Interfaces are needed to define the methods that are exposed by an SMC, the events that it can generate and which notifications it can receive. These aspects are discussed below.

2.2.1 Roles

Policies for the management of an SMC need to take into account possible interactions with other SMCs and apply to those SMCs when the interactions occur. For example, authorisations may need to be specified that give a remote SMC access to some of the interfaces and managed resources. To achieve this we define within an SMC placeholders for interacting SMCs; we refer to such placeholders as *roles* to which specific devices or external SMCs can be assigned depending on their capabilities and credentials. This enables policies relating to interactions or management to be defined in advance in terms of the roles, and applied to the appropriate SMC or device when they are assigned to that role. For example, a body area network for a patient may have roles defined for the nurse and doctor with which it would typically interact. Similarly a doctor's SMC can have a role for a patient to which the patient's SMC is assigned when the doctor visits the patient. This role allows the doctor to read specific sensors and adjust alarm thresholds within the patient's SMC so the interface to the patient's SMC must support this interaction.

2.2.2 Interface

An SMC specifies the functionality it provides to other SMCs through an interface. This must support both the *core* management functions of the SMC and also the management of application-specific behaviour (*customised* interfaces). An interface defines the operations supported by the SMC, as well as the events that can be sent and received to/from remote SMCs and effectively specifies its capability. Formally, an interface description is defined as:

$$Interface_i \ = \ \langle O, E, N \rangle$$

Where:

- O is a set of *operations*, which are methods the interface provides to remote SMCs;
- E is a set of *events*, which are generated (published externally) by the SMC (i.e. to which external SMCs can subscribe);
- N is a set of *notifications*, which are incoming events to which the SMC has subscribed.

Interfaces in the SMC model are specified in terms of the events that an SMC can send or receive and the operations it supports. Rather than using an *interface definition language* (IDL), SMC interfaces are described in Ponder2 notation using coding conventions to restrict the interface specification. Our work focuses on the management and adaptation interactions between SMCs and we do not seek to define general protocols for communication between arbitrary software components. Thus, SMC interfaces do not define request/reply protocols or sequences of events as typically found in WSDL descriptions [19], although application-specific protocols may be defined by an application programmer. SMC interfaces also do not currently implement *software contracts* [20] for validating SMC invocations through preconditions, postconditions and invariants that must be satisfied, although contracts are being considered as part of our future work.

The functions that an SMC exposes to another do not depend solely on the type of the SMCs but also on the specific instances concerned and the context for the interaction. For example the interfaces to a patient's SMC provided to a doctor may be different depending on the doctor's specialisation and whether the interaction occurs in the context of a hospital or in the street during an emergency. For this reason, SMCs expose to each other *customised interfaces* depending on the SMCs type, identity and context of the interaction. An SMC can support multiple customised interfaces, which allow different interacting SMCs to have a different view of the functionality the SMC exports.

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 would make all operations to services and resources visible externally even if they are not accessible. This may have both security implications, e.g., in terms of revealing capabilities of an autonomous system, and privacy implications, e.g., in e-health applications. Instead, the customised interfaces enable different SMCs to have different *'views'* of the functionality provided by an SMC.

2.2.3 Interaction Establishment

The establishment of an interaction is a three-step process: *(1)* a remote SMC that is capable of joining an interaction is dynamically discovered; *(2)* information about the discovered SMC is used to determine to which role the SMC will be assigned; and *(3)* the policies relating to that role are downloaded into the SMC. The first two steps of the process are detailed below, and the third step is discussed in Section 2.2.4.

Interaction establishment is initiated as a result of the discovery service of an SMC detecting the presence of another SMC in response to the *identity message* (id;type[;extra]), where *id* is the address of the SMC, e.g. *rmi://gumstix4.doc.ic.ac.uk/smc* if RMI communication is being used, *type* is the kind of SMC, e.g. *doctor* or *patient*, and *extra* is additional information about the SMC, e.g. a digital certificate. When a remote SMC is detected, the discoverer generates the event *found_SMC* within its local event bus [1]. The event contains information about the discovered SMC, and allows the components and services within the discoverer SMC to handle it as appropriate. In particular, the address of the remote SMC is used to obtain that SMC's interface.

The patient provides one interface to a doctor which is possibly different from the interface provided to a nurse. When an SMC provides an interface to another, this is pre-determined by the *type* information of both discoverer and discovered SMCs. In our implementation, local policies running in each SMC define which interfaces should be provided to other SMCs based on their types. An SMC may need to authenticate its partner using the *extra* information supplied in the identity message before handling a specific customised interface.

Each role specifies an *expected interface*, in terms of *operations*, *events* and *notifications*, that a remote SMC needs to satisfy in order to be assigned to that role. Formally, let $Intf_c = \langle O_c, E_c, N_c \rangle$ be the customised interface provided by a discovered SMC, let r be a role defined within the discoverer SMC, and let $Intf_r = \langle O_r, E_r, N_r \rangle$ be the expected interface for that role, then the assignment of the SMC which provides $Intf_c$ to role r is subject to the following condition being satisfied:

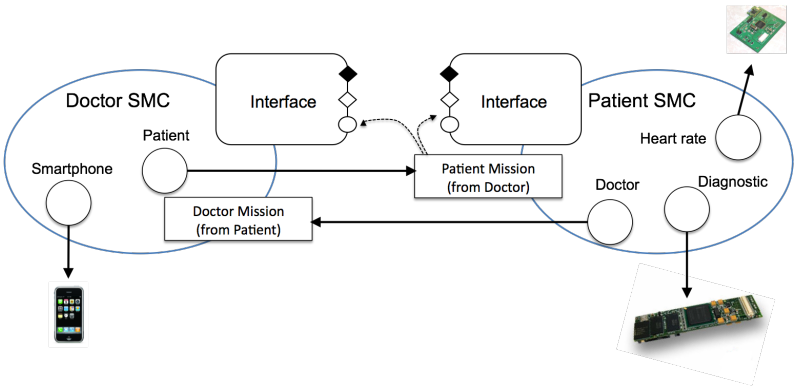


Fig. 2 SMC mission exchange between Doctor SMC and Patient SMC, allowing them to dynamically load management policies into each other [1]

$$assign(Intf_c, r) \rightarrow (O_r \subseteq O_c) \wedge (E_r \subseteq E_c) \wedge (N_r \subseteq N_c)$$

Thus policies can be written in terms of the functionality specified by the role's expected interface because any SMC assigned to the respective role must support at least that minimum functionality. This ensures that SMCs complying with a role's expected interface will be capable of executing the policies previously written for that role.

2.2.4 Missions

A *mission* is a set of policies which can be downloaded into an SMC assigned to a role. The policies specify how it should react to both internal events and external notifications by invoking management actions locally or on remote SMCs. Thus a mission provides the means for dynamically defining the behaviour of the assigned SMC in the context of the interaction. Missions are normally pre-specified by an application *'programmer'*. Roles and expected interfaces define a scope for specifying the policies contained in a mission. When a new SMC is discovered, missions defined within the discoverer SMC can be downloaded and instantiated on the discovered SMC. Mission downloading and instantiation are dependent on the existence of authorisation policies allowing one SMC to perform these actions on another.

Fig. 2 illustrates a mission exchange between a doctor and a patient SMC. When a doctor SMC discovers a patient's body-area network SMC, a mission is downloaded and instantiated on the patient device if permitted, e.g. for ECG monitoring relying on the sensors and devices available within the patient SMC. Similarly, the patient may also load and instantiate a mission at the doctor, defining the policies it expects the doctor to fulfil, e.g. for re-calibration of the patient's sensors. In essence, missions are a constrained form of programming a remote SMC. Before instantiating the mission and its policies, the receiving SMC must validate the mission to prevent it from compromising the integrity of the SMC. Mission specification and validation is described in great detail in [5].

3 Management Patterns for Building SMC Interactions

As previously discussed, the SMC is the basic autonomic building block component for large-scale policy-based systems. However, a large system may have many SMCs which need to interact. The disaster relief scenario indicated the need for cooperation between teams from paramedics, police and fire departments, possibly with help from local authorities and international charities. To cater for large-scale systems, it is necessary to compose and aggregate multiple SMCs into a single larger SMC component with a well-defined interface so that the composed SMC can be treated as a single management unit, hiding its internal complexity. Manually specifying complex policy-driven interactions in terms of the basic mechanisms discussed in the previous section is cumbersome and error-prone. Instead, support is needed for the design and establishment of policy-based interactions between SMCs that can be built in a systematic and reusable manner.

3.1 A Catalogue of Patterns for SMC Interactions

The commonality exhibited by many applications relating to management relationships, communication relationships and the structuring of the components can be exploited to simplify the design process by using predefined architectural patterns. We distinguish between *control patterns* that indicate which component is in overall control and capture task allocation strategies, *communication patterns* that define event forwarding strategies between SMCs, and *management structure patterns* that define the relationship between components, interface visibility and encapsulation criteria for the interactions. These can be seen as *complementary* dimensions for defining policy-based SMC interactions. They are complimentary in that control patterns support the exchange of policies, communication patterns support the exchange of events, which are required for triggering policies, and structural patterns support the exchange of interfaces, which are required for validating the actions prescribed by policies.

The concept of design patterns [21] has been used in software engineering for many years as a means of providing standard solutions to recurring problems. In our approach, each pattern defines the roles for the participants in the interaction and the policies governing the behaviour of the participants in the interaction. A library of patterns can then be specified and patterns can be instantiated dynamically to compose and federate SMCs into larger structures. This has similarity to the work on software architectures, which advocates the use of components and connectors as a means of structuring software development for distributed systems [22], although they do not cater for the use of policies or the adaptive behaviour needed for mobile ubiquitous systems. Furthermore, structural, control, and communication patterns can be combined into application-specific patterns e.g., for the health and social care workers involved in the care of a patient with a chronic disease who is living at home. Although a large number of different interactions can be defined, typically applications tend to use small subsets of these interactions. Table 1 presents a summary of the catalogue of architectural patterns for SMC interactions and these are elaborated below.

Table 1 Catalogue of architectural patterns for SMC interactions

Category	*Architectural Pattern*	*Description*
Management Structure	Peer-to-Peer Management	Ordinary, symmetrical mode of interaction between SMCs that exchange interfaces
	Management Composition	One SMC encapsulates another's interface and determines its visibility through mediation
	Management Aggregation	Inner SMC becomes resource of outer but without imposing encapsulation (allows sharing)
	Fusion	Combines the interfaces, policies, and managed objects of two constituent SMCs into a new SMC
Control	Hierarchical Control	One top-level SMC controls the execution of a set of leaf SMCs
	Cooperative Control	One leaf SMC is controlled by a set of cooperating manager top-level SMCs
	Auction	Task allocation employing a negotiation approach (issuers and bidders)
	Distributed Control	Fully decentralised interaction where SMCs can both load and receive tasks from their partners
Communication	Publish-subscribe	Provides a way of directly forwarding events to interacting SMCs
	Shared Blackboard	Provides a common means of sharing information among SMCs
	Diffusion	Propagation via intermediate nodes in a network
	Store-and-Forward	Useful in ad-hoc settings where SMCs do not have a permanent connection to their partners

3.1.1 Management Structure Patterns

Structural patterns define how SMCs are organised with respect to the access of their interfaces. We strictly focus on management structures for organising SMCs rather than on general software design patterns for component or service composition. We used the terms *management composition* and *management aggregation* to distinguish them from the composition and aggregation designs that are traditionally used in the software engineering community. Management structures provide abstractions for encapsulation, mediation and mapping of interfaces, and also for combining two or more constituent SMCs to form a new SMC. To some extent, data access and transformation patterns can also be supported by structural patterns through interface mediation. Structural patterns do not preclude the use of other patterns, e.g., to define the internal structure of a composition, or to define control or communication patterns between the SMCs.

Management Composition is an interaction in which an outer SMC encapsulates inner SMCs which are effectively its internal resources or managed objects. The benefit of composing a number of smaller SMCs into a larger one is to reduce the management complexity, encapsulating the policies and events required to manage SMCs within a limited scope. Any interaction with an internal SMC from the outside is *mediated* via the outer SMC interface. In this case an SMC can only be a member of a single composition – thus a composed SMC does not respond to other discovery requests. Composed components may be independent devices, communicating over a network. If internal components fail, interaction with the external world may temporarily fail but it is possible to have back-up internal in-

terface components which take over the internal role (not currently implemented) to make failure transparent to the external world. The healthcare body sensor network is an example of a composite SMC as the patient's controller mediates all interaction with the external world and may propagate events into or from the BSN sensors for interactions with healthcare workers, or the environment. Body sensors must be prevented from interacting with another nearby patient's body network e.g. in a hospital or in a public place. Security mechanisms such as access control, authentication and encryption are needed to enforce this encapsulation as described in [13,14]. Destroying a composite SMC would typically destroy the management relationship across all internal components, but it does not imply destroying the managed components themselves.

Peer-to-Peer management relationships are used to organise interactions in which the components are equals, without encapsulation or interface management i.e. no single component is in overall control. The exchanges of interfaces occur in a symmetric manner and an SMC may take part in multiple peer-to-peer relationships. Each peer will have a role relating to the other. Peers do not really manage each other but they may interact in a client-server or collaborative pattern relevant to the application. Peer-to-peer management is exemplified by the interaction between the doctor's and patient's mobile devices. For example, in Fig. 2 both doctor and patient SMC have remote role placeholders and can exchange interfaces. Components within a management composition may form a peer-to-peer management structure, but would have to be informed about their peers as they are not discoverable.

Aggregation management is a 'looser' form of combining SMCs into a group than composition. It facilitates the hierarchical structuring of SMCs but does not imply mediation or encapsulation. Aggregated components are visible (i.e. discoverable) from outside the SMC and may be partly managed by external components. In the disaster relief scenario, the SMCs representing the various organisations are forming an aggregation. The SMCs may actually be managed by other SMCs representing their parent organisations. Aggregation can be used to form hierarchical relationships between sets of SMCs – lower level SMCs provide services to higher level ones which combine these primitive services into higher level ones offered to clients. For example, environmental monitoring sensors on cars could form a mobile ad-hoc network for reporting readings to fixed data receivers around the city. A web-based pollution service would query these data loggers to analyse and provide a service for users to check pollution levels via smartphones or people with specific sensitivity could subscribe to an SMS messaging service to receive warning of high pollution levels in specific areas. Thus there are four levels of service hierarchy in this aggregated application – mobile users, web-service, fixed data loggers and mobile sensors.

Fusion merges two or more SMCs to form a single SMC. For example two independent teams of robots, each with their own commanders may merge to form a single team with one commander for the purposes of a particular mission.

3.1.2 Control Patterns

Control patterns capture task-allocation strategies and control aspects. While missions define what policies are being exchanged, control patterns specify how

these exchanges occur. Control patterns also assist in the specification of manager/managed relationships between SMCs, facilitating the deployment of the authorisation policies which are required for mission downloading. Control patterns are complementary to structural patterns: for example, the SMCs forming a composition or an aggregation structure may rely on different control patterns for defining their task-allocation and policy-exchange behaviour.

Hierarchical Control exists when one SMC is responsible for managing another as is the case between the outer SMC and inner SMCs in a composition. A manager typically provides tasks and policies to subordinate components. For example, in the body sensor network scenario, the patient's controller is a smartphone with more powerful resources than the sensor nodes and 3G or Wi-Fi communication links for interacting with the external world. There is a hierarchical relationship between the controller and the body nodes and the controller will allocate policies to the nodes. However hierarchical control does not imply management composition for all applications. The disaster relief scenario may have a central command centre with a hierarchical control relationship to the collaborating organisational SMCs but there is no composition.

Cooperative Control occurs when a set of cooperating manager SMCs control the execution of a subordinate SMC by delegating policies to it. This implies that the manager SMCs have *rights of programmability* over the subordinate. These multiple managers may cause conflicting policies to be loaded, which must be addressed by mechanisms such as prioritisation of policies or use of application-specific meta-policies. For example, a shared robot may be subject to policies downloaded from two cooperating teams using a cooperative control relationship.

Auction-based Control in which SMCs bid for tasks to perform on behalf of managers might suit some applications where processing is off-loaded from portable devices to more powerful ones within the infrastructure, or when tasks are distributed to a collaboration of robots that have differing capabilities.

Distributed Control occurs when a number of SMCs cooperate to achieve an overall goal by exchanging tasks and policies with each other but there is no subordination relationship. This obviously maps onto a peer-to-peer structural relationship, but elements in a peer-to-peer relationship may only exchange data and not management information, so do not necessarily form a distributed control relationship.

3.1.3 Communication Patterns

Communication between SMCs typically occurs through asynchronous event exchanges, as events are required for triggering policies running on remote SMCs. Communication patterns specify how events are exchanged and how event buses of various SMCs are interconnected.

Publish-Subscribe is the primary means for disseminating events within the SMC as well as between SMCs in composition, aggregation and peer-to-peer structures. A Publish-subscribe system is particularly useful for disseminating management and context information. This may be extended from simple events to combining multiple correlated event sequences to form more complex events [23].

Shared Blackboard is the traditional pattern where all participants communicate by writing information into a shared knowledge base. Different implementa-

tions are possible depending on whether this shared knowledge base is replicated to each of the participants, centrally maintained, etc. For example, this pattern could be used as a means of sharing structured information such as casualties, resources available or current search areas in the disaster scenario.

Diffusion is used in sensor networks with sensor readings and events propagating via intermediate nodes to one or more sinks where the data can be analysed and aggregated. Algorithms for epidemic data dissemination in federated system [24], possibly with built-in fault-tolerance mechanisms, could also be implemented.

Store-and-Forward is often used in delay tolerant ad-hoc networks. For example, environment monitoring sensors could use this pattern for collecting data and subsequently delivery to remote logging servers when network connectivity is available.

The above are not meant to be an exhaustive list of possible patterns for designing ubiquitous systems. We have indicated some of the generic patterns, but there can be many different application-specific ones. Finally, the categories (structural, control and communication) of architectural patterns are certainly not all-inclusive. Instead, the categories above were identified through our need of building policy-based interactions between SMCs. It is likely that new categories, e.g., security patterns, will be added to the taxonomy presented in this paper as we apply the pattern abstraction to a broader spectrum of autonomous systems.

3.2 Engineering Policy-based SMC Interactions

The control, communication and structural patterns can be seen as complementary perspectives for defining policy-based SMC collaborations. Each architectural pattern defines a particular abstraction and can be used to specify independently the different aspects of an interactions in terms of: *(a)* the *exchange of policies*; or *(b)* the *exchange of events* for triggering policies; or *(c)* the *exchange of interfaces* for validating the actions prescribed by policies. Based on the catalogue presented above we now define a methodology for engineering policy-based SMC interactions, which relies on the combination of patterns as design elements of a collaboration.

A pattern defines its own set of specific roles, i.e. the placeholders for the participants in the interaction, according to the abstraction it enforces. These *pattern-specific roles* are associated with the *domain roles*, e.g., *patient* and *sensor*, through a process called *binding*. Therefore, patterns enforcing specific semantics for event exchanges between SMCs, e.g., a *diffusion*, will have their pattern-specific roles, e.g., *source* or *target*, bound to the domain roles in an SMC. These bindings define how the SMCs that will be assigned to domain roles should execute their event exchanges. Similarly, this is used to specify how SMCs execute their policy downloading through control patterns, e.g., *hierarchical* or *auction*, and how SMCs organise themselves in terms of interface access using structural patterns, e.g., *peer-to-peer* or *composition*. Furthermore, each pattern defines a specific algorithm or protocol for achieving the exchange of policies, events or interfaces, with well-defined properties. The binding of pattern-specific roles to domain roles permits the design of complex policy-based interactions incrementally. The pattern-specific roles bound to a domain role ultimately dictate how an SMC will behave: as an *outer*/*inner* w.r.t a *composition*, as an *issuer*/*bidder* w.r.t. an *auction*, or as a

source/*target* w.r.t. a *diffusion* pattern. A more exhaustive catalogue of patterns for SMCs has been presented in [25].

We distinguish between specification and instantiation of an interaction. The *specification* consists of binding architectural patterns to roles in the local domain of an SMC. This defines how SMCs that will be assigned to these roles are expected to establish their interactions. When actual SMCs are assigned to domain roles, the patterns which were previously bound will be *instantiated*, and interactions will be established in the form of exchanges of policies, events and interfaces as prescribed by the patterns.

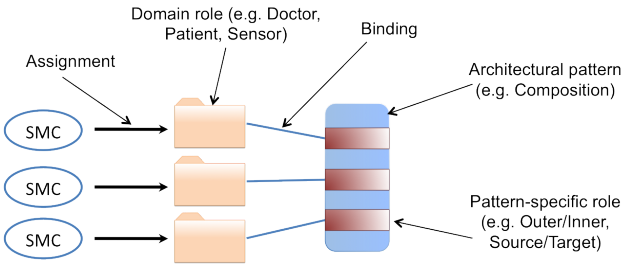


Fig. 3 Architectural patterns, bindings and roles

The relationship between domain roles in an SMC, patterns and pattern-specific roles is illustrated in Fig. 3. For a given pattern, e.g. *composition*, pattern-specific roles, e.g. *outer* and *inner*, are *bound* to roles in the SMC's domain, e.g. *doctor*, *patient*, *sensor*. This defines how the SMCs that will be assigned to the roles are expected to establish their interactions.

A pattern implicitly specifies a set of *requirements* that are inherited by the roles to which the pattern is bound. SMCs assigned to these roles must provide an interface which can support the requirement. For example, a pattern that specifies the forwarding of the event *heart_rate* from one SMC to another will typically require this event to be emitted by the *source*'s interface and defined as incoming at the *target*'s interface. Thus a *requirement* is a function which associates a pattern-specific role with a number of operations, events or notifications which must be supported by the SMC assigned to that specific role:

$$requirement : patternRoles \rightarrow (O \cup E \cup N)$$

Domain roles in an SMC aggregate both the *behaviours* and *requirements* of all pattern-specific roles which were bound to them. For example, a given domain role can be simultaneously bound to an *inner* (through a *composition*) and to a *source* (through a *diffusion*) pattern-specific roles.

A *pattern* is defined as a set of pattern-specific roles, an associated behaviour and a set of requirements:

$$pattern = \langle PatternRoles, Behaviour, Requirements \rangle$$

where *PatternRoles* is the set of roles defined by this pattern, *Behaviour* is the implementation-specific behaviour defined by the pattern, and *Requirements* is the

set of requirements (in terms of operations, events and notifications) that must be satisfied by each participant in order to accomplish the behaviour prescribed by the pattern.

An *interaction specification* that is enforced by an SMC consists of a number of domain roles, architectural patterns, and how they are bound to each other within the SMC:

$$specification = \langle Roles, Patterns, Bindings \rangle$$

where *Roles* is the set of roles defined in the local domain of the SMC enforcing this specification, *Patterns* is the set of architectural patterns bound to these roles through the set of bindings *Bindings*.

A *binding* of a pattern with respect to an interaction specification associates each pattern-specific role defined in the architectural pattern with a role within the SMC's domain. Hence:

$$binding(pattern, spec) \rightarrow$$
$$\forall x \in patternRoles_{pattern}, \exists y \in Roles_{spec} : \; y := y \; \odot \; x$$

where the $\odot$ operator is defined as follows. Let $\odot$ be a binary operator, when applied to a pattern-specific role (right operand) and a domain role (left operand) it performs the union ($\cup$) of the set of behaviours associated with both operands, and the union of the set of requirements that is also associated with both operands. The resulting sets are then assigned to the domain role. Assigning a larger set of requirements to a domain role means adding new restrictions to the *expected interface* of that role, which will have to be satisfied by the SMC assigned to it, as discussed in Section 2.2.3.

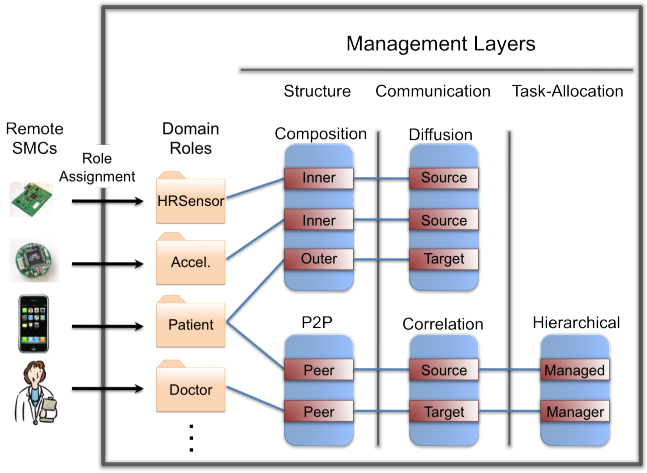


Fig. 4 Composition model: (1) patterns are bound to *roles* in the local domain; (2) remote SMCs are assigned to these roles; (3) patterns are instantiated and the behaviour associated with each pattern is enforced in the SMCs

When remote SMCs are discovered at run-time, they will be *assigned* to roles in the domain of the discoverer SMC if they satisfy the requirements for these

roles, and the patterns which were previously bound will be instantiated. These will dictate how these SMCs should interact with each other (Fig. 4). Hence, an *assignment* of an SMC to a domain role within a specification will cause all patterns bound to this role to be instantiated in the SMC:

$$assignment(SMC, role_{spec}) \rightarrow \\ \forall pt \in patterns_{spec} : \forall x \in patternRoles_{pt} : x \odot^{-1} role_{spec} \\ \rightarrow instantiate(SMC, pt, x)$$

where the $\odot^{-1}$ operator is defined as follows. Let $\odot^{-1}$ be a binary operator, it evaluates to *true* if a pattern-specific role (left operand) and a domain role (right operand) are *bound*. This causes the instantiation of the respective architectural pattern. The *instantiation* of a pattern is defined by the behaviour associated with the pattern, in terms of how the exchange of interfaces, policies or events is achieved. This operation takes as arguments an SMC, the pattern to be instantiated and the specific role within the pattern that this SMC will be fulfilling.

This model allows us to define independent layers of management for policy-based SMC interactions, where the structural, communication and control aspects can be specified by reusing common abstractions expressed as architectural patterns. There are dependencies among the patterns: *structural patterns* must be instantiated first, as they enable the exchange of customised interfaces, e.g. *doctor* interface, *patient* interface; *communication patterns* are then instantiated to define patterns in terms of the events provided by these interfaces; *control patterns* must be the last, as the policies downloaded depend both on the operations provided by the interface, as well as on the events forwarded by a communication pattern. The consistent use of patterns is discussed in Section 4.

3.3 Healthcare Scenario Revisited

Consider some of the requirements for a typical healthcare monitoring application using the personal SMC representing a patient's *body-area network* described in Section 1. A doctor or nurse SMC running in his/her smartphone must interact with patients, downloading monitoring tasks and collecting monitored results. Monitoring tasks running on the patient body-area network allow the patient to be self-monitored in his own home environment, thus promoting reduced usage of hospital resources and better medical evidence data for the clinical condition and its treatment. Tasks loaded by the healthcare worker continually run on the patient's SMC, relying on information provided by his body sensors. Two situations regarding the collected data are of interest: *(a)* under normal circumstances data may be stored on a home server SMC, for synthesis and for subsequent delivery to the GP surgery (e.g. a scheduling task that sends a subset of this data every seventy-two hours); and *(b)* in an emergency situation, data is used to request immediate assistance (e.g. if monitored information indicates a heart attack might be imminent).

Fig. 5 outlines the SMCs involved in this scenario. The informal description does not specify how their relationships are realised. Although the *Self-Managed Cell* is suitable for representing autonomous components, we still need adequate abstractions for expressing their interactions. For example, if multiple SMCs are

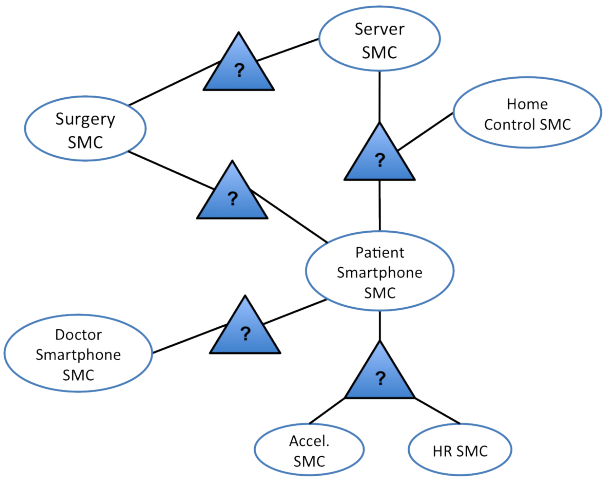


Fig. 5 Healthcare monitoring scenario: each triangle represents an interaction that requires a different combination of management abstractions

used on a patient to monitor related conditions they would typically be composed in a single autonomous SMC. In contrast interactions between the patient's body area network and healthcare personnel would typically be interactions between peers. Collected data is forwarded differently in a body-area network or in a home environment. Finally, tasks for physiological monitoring or data synthesis are loaded in specific situations and subject to different conditions. The purpose of architectural patterns is to provide a systematic way of designing interactions and clearly representing their management layers.

Fig. 6 shows a graphical representation of the scenario where abstractions were chosen for interface, event and policy exchanges. Each of the five columns in Fig. 6 corresponds to one of the triangles in Fig. 5. The graphical representation shows that the patient smartphone and the two sensor SMCs (heart-rate and accelerometer) are bound through a *composition* structural relationship as *encapsulation* of the sensor SMCs is needed to prevent interaction with other nearby patient body networks. Although the resources are encapsulated, operations for reading sensor measurements are typically mapped to the patient's interface. Sensors forward their events (e.g. the heart rate goes above a certain threshold) to the patient smartphone SMC through a *diffusion* event forwarding, where each sensor plays the *source* role while the patient smartphone plays the *target*.

The interaction between patient and doctor has different requirements. An encapsulation would not apply, as a patient may interact with multiple doctors, and vice-versa. A *peer* abstraction is more suitable in this case, which leads to a simple exchange of interfaces but with no additional mapping or encapsulation. In terms of task-allocation, a doctor will typically specify policy downloading strategies into the patient through a *hierarchical control* pattern. The tasks downloaded in the form of *missions* are defined in the pattern parameterisation (e.g. an ECG monitoring mission). Similarly, the conditions when such tasks must be downloaded are also part of the pattern parameterisation. The example can be further elaborated with the specification of the remaining interactions with the *home* and

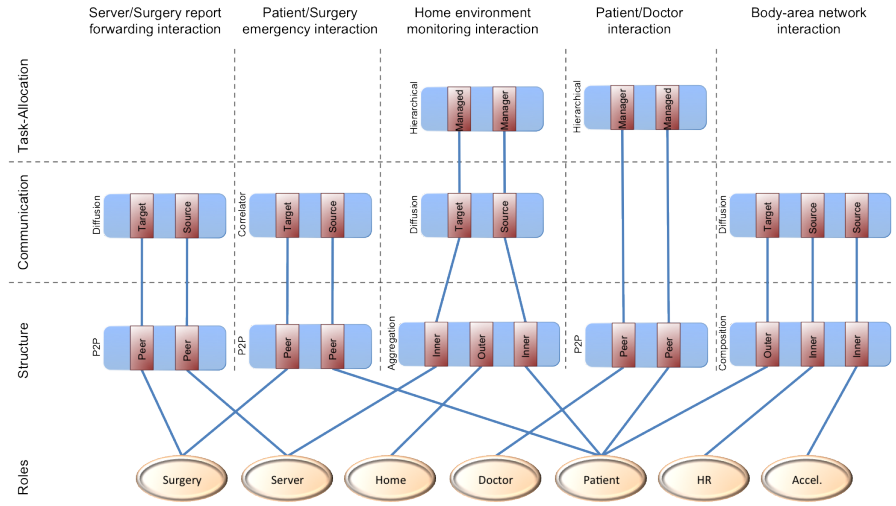


Fig. 6 Architectural representation of the health monitoring scenario: the illustration shows the overall configuration of the collaboration with the selected abstractions

surgery SMCs. Details about the implementation of architectural patterns will be presented in Section 5.

4 Formal Specification and Model-Checking

Consistent policy deployment is crucial as often SMCs form autonomous administrative domains. When these SMCs are composed or federated, inconsistencies, conflicting policies or interface incompatibility between the devices available may prevent them from operating as originally expected. The definition of a formal model assists in the design of SMC collaborations and allows the verification of the correctness of anticipated interactions before these are implemented or policies are deployed in physical devices, e.g. smartphones, sensors, or network equipment.

We chose the *Alloy Analyzer* [26,27] as the platform for the formal specification of the SMC behaviour. *Alloy* is a *declarative* modelling language based on first-order logic and used for expressing complex structural constraints and behaviour in a software system. It differs from pi-calculus [28], ambient calculus [29] and channel ambient calculus [30] which model the computation *operationally*. We found *Alloy* more natural and concise for describing SMC interactions and the integrity constraints related to SMC management. Alloy is similar in spirit to other formal specification languages such as Z [31], VDM [32] and B [33] but, unlike those, is capable of fully automatic analysis in the style of a model-checker[4].

Models written in *Alloy* are automatically checked for correctness using the *Alloy Analyzer*. The analyser performs a finite scope check, i.e. analysis is performed over restricted scopes on the number of objects (instances) to be used, which is defined by the user (the user-specified scope makes the problem finite and thus

[4] http://alloy.mit.edu/alloy/

reducible to a boolean formula). This is based on the *small scope hypothesis* [27], that for any flawed design a counter-example should be found by an exhaustive search within a comparatively small, bounded scope. Defining a formal Alloy specification for the behaviour of SMCs enables: (1) formally capturing the static and dynamic aspects of the structure and behaviour of their interactions; (2) automatically verifying the consistency of SMC collaborations by using its analyser; (3) simulating SMC behaviour in complex interactions. The toolset also provides a visualisation tool which can be used to display examples or counter-examples graphically (figures in this section were generated by this visualiser, with small hand edits of names to aid comprehension).

4.1 Basic Self-Managed Cell Model

An *Alloy* model consists of a set of *signatures* and a set of *predicates*. Signatures define the structure of the model, where each signature represents a concept in the model and its relationship to other concepts. Predicates define the behaviour of a model, showing what properties hold before and after the execution of an operation. Predicates can be used to indicate the changes that happen when an SMC is discovered, when an SMC departs, when an SMC is assigned to a role, etc.

We initially defined the basic concepts such as SMC, Role, Interface (containing Events, Operations and Notifications supported by that interface), and the operations for discovery of a new SMC, departure of an SMC, role assignment and role de-assignment. For example, an SMC provides one or more interfaces. The signature *Interface* defines the operations (methods that can be invoked), the events (which can be published externally) and the notifications (which are external events of which the SMC can be notified) supported by that interface. *Interface* is an **abstract** signature, which means it can be extended to define specialised interfaces for different applications.

```
abstract sig Interface
{
    operations: set Operation,
    events: set Event,
    notifications: set Notification
}
```

Simple predicates were defined to cater for the specification of discovery and departure of SMCs, and assignment and de-assignment of SMCs to/from roles. These, however, merely serve as the foundation for building more complex configurations, which rely on policy-based interactions and different management relationships that are encoded as architectural patterns. These concepts are defined in the following.

4.2 Policies

We now discuss the specification of obligation and authorisation policies, and the predicates related to policy downloading. In particular, the signature *ConcreteObligation* defines the subject and target roles for a policy, the event that triggers the policy and the action to be invoked in response.

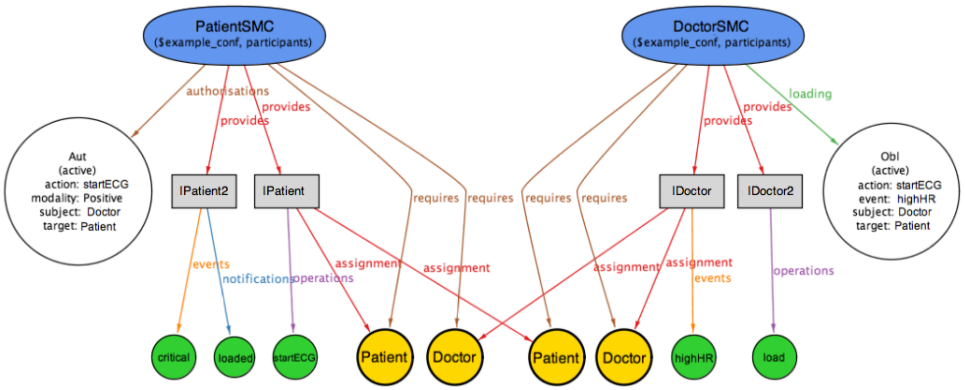
```
sig ConcreteObligation extends Obligation
{
      subject: one Role,
      event: one Event,
      action: one Operation,
      target: one Role
}
```

Similarly, the signature *ConcreteAuthorisation* defines a subject role, a target role, an action and the modality of the policy (which can be either positive or negative).

```
sig ConcreteAuthorisation extends Authorisation
{
     modality: one Modality,
     subject: one Role,
     action: one Operation,
     target: one Role
}
```

The formal specification of policies in Alloy enables several types of analysis, including policy conflict detection. Our focus, however, is not on the ability to detect policy conflicts, but instead on the verification of SMC collaborations and whether their participants are able to enforce the policies. For example, it is possible to verify whether the SMCs assigned to roles are capable of enforcing the policies associated with those roles, or whether all obligations enforced by collaborating SMCs have a corresponding authorisation policy. Consider the interaction illustrated in Fig. 7(a). *DoctorSMC* enforces the obligation *Obl*, which states that the subject role (*Doctor*) must invoke on the target role (*Patient*) the action *startECG* in response to the event *highHR*. The interface *IDoctor* provided by *DoctorSMC* is locally assigned to the subject role, and interface *IPatient* provided by *PatientSMC* is assigned to the target role in *DoctorSMC*. Moreover, *PatientSMC* enforces authorisation *Aut*, which states that the subject role (*Doctor*) is allowed to invoke action *startECG* on the target role (*Patient*). The remote interface *IDoctor* provided by *DoctorSMC* is assigned to the *Doctor* role in *PatientSMC*, whereas the local interface *IPatient* is assigned to the *Patient* role, which is also the target of the policy being enforced by this SMC.

Verifications are specified as *predicates* in the Alloy model, which define properties to be checked for a specific *configuration* (a configuration is an additional signature representing the current state of an interaction). For example, the role assignment can be type-checked to verify whether the interfaces assigned to roles satisfy the requirements for the policies associated with those roles. The assignments given in the example above satisfy this property, as interface *IDoctor* (which is assigned to the *Doctor* role in *DoctorSMC*) supports the event *highHR* (which is required for triggering the obligation policy). Similarly, interface *IPatient* (which is assigned to the *Patient* role in *PatientSMC*) supports action *startECG* (which is the action to be allowed execution by the authorisation in that SMC). Each obligation has a corresponding authorisation policy as validated by the *checkAuthObl* predicate in Fig. 7(b), which verifies that *for all* obligations there is *some* authorisation *such that* the SMC assigned to the obligation's subject (resp. target) is the same assigned to the authorisation's subject (resp. target), and both policies refer to the same action. More traditional types of analysis, such as modality or application-specific conflicts are defined in a similar manner.



(a)

```
pred checkAuthObl[conf:Configuration]
{
  all smc1:conf.participants, obl:(conf.active & ConcreteObligation)
  {
    obl in (smc1.obligations + smc1.(conf.loading)) ⇒
        some smc2:conf.participants,
             aut:(conf.active & ConcreteAuthorisation)
        {
              (aut in smc2.authorisations)
          and (aut.modality in Positive)
          and (obl.action == aut.action)
          and ((obl.subject).~(conf.assignment)
                  == (aut.subject).~(conf.assignment))
          and ((obl.target).~(conf.assignment)
                  == (aut.target).~(conf.assignment))
        }
  }
}
```

(b)

Fig. 7 Alloy graphical representation of a policy configuration between SMCs (a), and *checkAuthObl* predicate used to verify that for each obligation there is an authorisation policy (b)

4.3 Architectural Patterns

We now define the architectural aspects of an interaction. In particular, an architectural pattern defines a set of pattern-specific roles (e.g. a *Composition* defines the roles *Outer* and *Inner*, a *Diffusion* defines the roles *Source* and *Target*, a *HierarchicalControl* defines the roles *Managed* and *Manager*, etc). These patterns can be combined, arranged in a particular manner, into higher-level application-specific patterns (e.g. a *body-area network* will often have a combination of structure, task-allocation and event forwarding aspects).

Two distinct steps related to SMC interactions are of particular importance: *(1) interaction specification* and *(2) interaction enforcement*. Interaction specification is defined by the bindings between pattern-specific roles and the roles required by an SMC – this defines how SMCs assigned to these roles will be expected to behave. Interaction enforcement is the instantiation of patterns bound to a set of roles, when actual SMCs are assigned to these roles, causing a new behaviour

to be added to the interaction in the form of a forwarding of events, mapping of interfaces or exchange of policy.

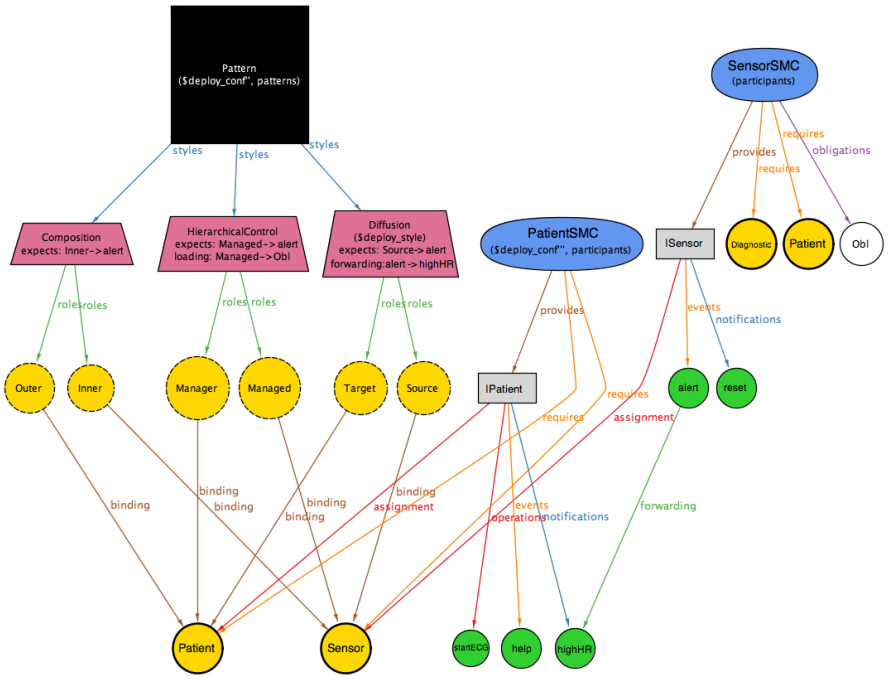


Fig. 8 Alloy graphical representation of an architectural configuration using three patterns to specify the interaction between *Patient* and *Sensor* roles

Fig. 8 shows an architectural configuration obtained from the Alloy model. In this example, *PatientSMC* requires two roles, *Patient* and *Sensor*, and these roles are bound to a number of pattern-specific roles. The *Sensor* role is bound to *Source* (through a *Diffusion*), *Managed* (through a *HierarchicalControl*) and *Inner* (through a *Composition*). Similarly, the *Patient* role is bound to *Target* (through a *Diffusion*), *Manager* (through a *HierarchicalControl*) and *Outer* (through a *Composition*). This architectural configuration means that whichever SMCs are assigned to the *Patient* and *Sensor* roles, they will behave according to each pattern bound to their roles. Each pattern specifies *(a)* what it *expects* from the SMCs that will be assigned to the roles, as well as *(b)* the *behaviour* to be added to the interaction after the pattern is instantiated. The example illustrates the instantiation of a *Diffusion* (the pattern is labelled as *"$deploy_style"*). This pattern is marked as *"expects: Source → alert"*, which states that, for whichever SMC is performing the *Source* role, its interface must provide the *alert* event. By following the binding of the pattern-specific role *Source* to the role *Sensor*, observing that interface *ISensor*, which is provided by *SensorSMC* is assigned to this role, we can see that this interface indeed provides the *alert* event, thus satisfying the requirements of the pattern. Similarly, this pattern is also labelled as *"forwarding: alert → highHR"* (this corresponds to the behaviour that must be added to the

interaction). The deployment of this behaviour can be seen through the arrow labelled *"forwarding"* between the *alert* event (provided by interface *ISensor*, which is assigned to the role *Sensor*) and the *highHR* notification (provided by *IPatient*, which is assigned to the role *Patient*). The behaviour added by the instantiation of other types of architectural patterns can be shown in a similar manner, in the form of mapping of interfaces, forwarding of events or downloading of policies.

This model can be used to check whether all the SMCs enforcing policies have the required events forwarded to them as well as whether all SMCs have access to the interfaces required for validating the actions prescribed by policies. These verifications increase the confidence in the robustness of policy-based interactions, as the interactions can be rigorously verified prior to deployment in actual devices.

5 Implementation and Evaluation

In the following we present details about our prototype, based on the Ponder2 framework, which was implemented in order to demonstrate the feasibility of the model and test its applicability. We also present an assessment of the model with respect to scalability and reusability aspects, as well as the performance of the prototype in devices with limited computational power and memory.

5.1 Prototype Implementation

In order to specify an SMC collaboration for a given purpose, e.g. healthcare monitoring, an application-programmer can rely on: *roles* that define the placeholders for actual SMCs, a repository of *management policies* pertinent to a particular scenario, and a repository of *architectural patterns* that provide reusable abstractions to define how policies, events and interfaces are exchanged. This *specification* is then given as input to the formal Alloy model for analysis. If the specification is successfully validated, it is ready for deployment in physical devices. Upon receiving an interaction specification, a device will locate the necessary SMCs, instantiate the patterns and establish an interaction among a group of SMCs. Sub-patterns in this specification can be further re-deployed in other SMCs, which will be responsible for enforcing different parts of a large interaction. This process is illustrated in Fig. 9(a). It would be possible to use the formal model to re-check the interaction during runtime, e.g. if a sensor fails, to ensure the policies can still run, however the use of model-checking in our implementation has been limited to design-time checks.

The different SMCs responsible for instantiating parts of an interaction or simply participating in an interaction instantiated by another SMC must run the *SMC runtime*. The SMC runtime extends the Ponder2 interpreter to facilitate SMC interactions (Fig. 9(b)). The standard functionality provided by Ponder2 implements a *discovery service*, which permits the SMC to advertise itself to both devices and other SMCs, an *event bus*, which supports the underlying event-based infrastructure within the SMC, and the *policy service* itself, which allows the specification and enforcement of both obligation and authorisation policies. These enable the basic functionality of the SMC as a feedback control-loop. Ponder2 also provides a *command interpreter*, which allows *PonderTalk* commands to be sent

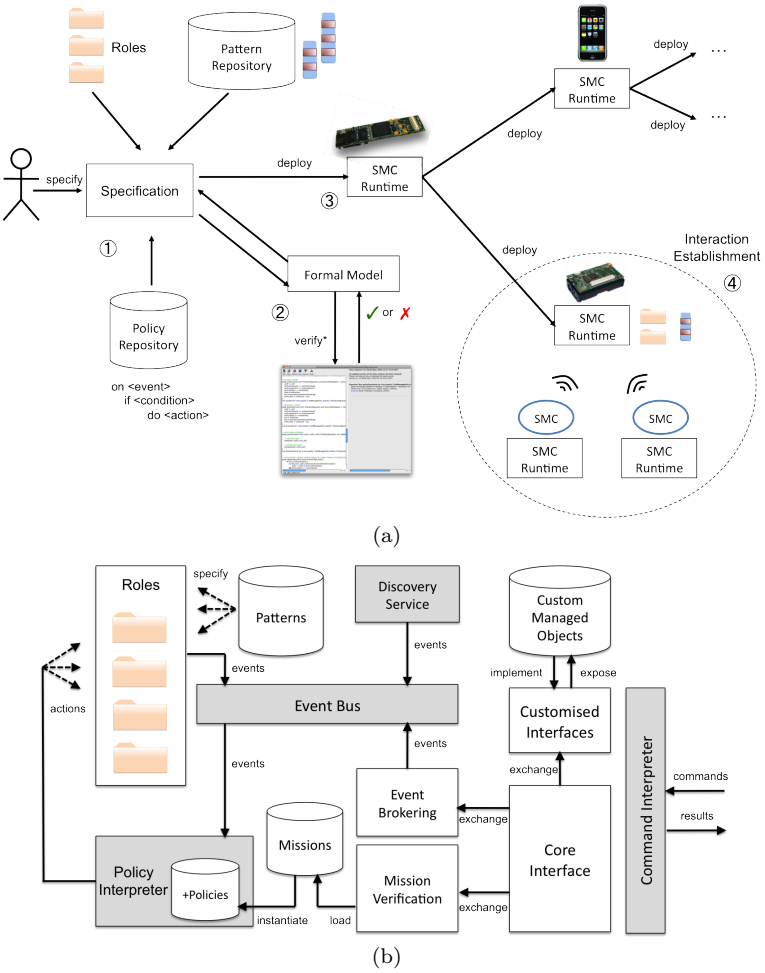


Fig. 9 (a) Specification and establishment of SMC interactions (b) SMC runtime (shaded blocks are standard Ponder2 components)

to configure and control the Ponder2 system. The example policies in Fig. 10 were specified in Ponder2, and they correspond to the policies presented more abstractly earlier in Section 2.1.3. Policies are applied to *managed objects* (*MOs*), which are stored in a local domain that implements a hierarchical namespace within Ponder2. The first policy is an obligation which is triggered by a heart rate event producer, and reconfigures the monitoring frequency of the oxygen saturation device, whereas the second policy is an authorisation required to permit management of the oxygen saturation device. Policies are also managed objects themselves, which means that policies can specify actions to be performed on other policies, e.g., having a policy that enables or disables another policy.

Our framework for SMC interactions adds a number of extensions to this infrastructure: a *core interface* enables the exchanges of policies, events and interfaces between SMCs. The implementation of its functionality relies on *managed objects* that implement parsing of missions and their verification, and brokers that allow

```
//Obligation policy to reconfigure the monitoring frequency
//of the oxygen saturation (OS) device
root/event at: 'HR' put: (root/factory/event create: #('value')).
root/policy
        at: 'myOblPolicy' put:
        ( p := root/factory/ecapolicy create.
          p event:     root/event/HR;
            condition: [ :value | value > 100 ];
            action:    [ root/OS setFreq: 10min ].
        ).
root/policy/myOblPolicy active: true.

//Authorisation policy required to permit management of
//the oxygen saturation (OS) device
newauthpol := root load: 'AuthorisationPolicy'.
root/factory at: 'newauthpol' put: newauthpol.
root/authdomain
         at: 'myAuthPolicy' put:
         ( newauthpol subject: root/patient
                      action: 'setFreq:'
                      target:  root/OS
                      focus:   'st'
         ).
root/authdomain/myAuthPolicy active: true.
```

Fig. 10 Examples of obligation and authorisation policies for healthcare monitoring and patient recovery specified in Ponder2 syntax

the subscription and forwarding of events between remote SMCs. A particular SMC can also have a dynamic set of application-specific *managed objects* that implement non-standard functionality, e.g. adapters for local sensors, authentication algorithms, etc, and this functionality is made available to remote SMCs via pre-specified *customised interfaces*. *Roles* are defined as placeholders in the domain structure provided by Ponder2, and we implemented syntactic verification between a role's expected interface and the SMC's provided interface before assigning SMC's to roles. Finally, a library of reusable architectural patterns enables the systematic specification of how a group of roles must interact.

The implementation of patterns is inspired by a well-known technique for implementing layered object-oriented design, called *mixin layers* [34]. Mixin layers are presented as a programming artefact, to specify a *collaboration* between a set of *classes* that are assigned to a set of *roles*. To some extent, *collaborations* in the mixin layer model can be seen as patterns in the SMC model − i.e. each pattern defines a number of roles, and SMCs play different roles in multiple patterns. Patterns are implemented as *managed objects* in Ponder2, and we defined a library of patterns divided into structural, task-allocation and communication categories. Each architectural pattern MO specifies a given algorithm or protocol for the exchange of policies, events or interfaces between a group of two or more participants. Pattern-specific roles define what parts of the algorithm or protocol each participant will be responsible for executing. This is achieved by ensuring that fragments of Ponder2 code (defined in the pattern MO) are executed by the SMCs when they are assigned to the respective roles. The implementation of patterns was presented in greater detail in [25].

5.2 Reusability and Scalability of SMC Interactions

Evaluating the support for scaling-up SMC interactions includes aspects such as reuse of code and ease in rapidly instantiating different types of interactions. Our experience in developing policy-based SMC applications shows that the use of architectural patterns satisfies these two issues. The parameterisation and instantiation of an individual architectural pattern typically requires about 10 lines or less of *PonderTalk* code. For comparison, the same interaction written manually without the aid of patterns would require about 30 lines of code. This is a factor of 3 increase, and for an application containing 100 of such interactions, that is 3,000 instead of 1,000 lines. An interaction for the exchange of interfaces, events or policies between a set of SMCs can be set up using an architectural pattern, by instantiating a single managed object which encapsulates the required support. For example, manually setting an interaction such as an event sharing scheme similar to the *SharedBlackboard* pattern, using only primitive abstractions not only requires a considerable amount of code to be written, but it is also error-prone: this is because the programmer is responsible for using the primitive commands for correctly setting up policies for event forwarding and installing the required event templates in separate locations − through the use of patterns, the pattern itself is responsible for enforcing the semantics for a given interaction instead of relying on the programmer to use the primitives appropriately.

The use of patterns also reduces the complexity and size of the interactions, by structuring and decreasing the number of necessary bindings between SMCs. A comparison can be made between abstractions for structuring an interaction; between peer-to-peer collaborations and compositions. Indeed, one of the motivations for compositions is to hide the complexity of large SMCs that comprise a set of smaller, yet autonomous, components, e.g. a body-area SMC. The number of interface exchanges for completely unstructured interactions, e.g. peer-to-peer, assuming that full connectivity is applied is given by the formula

$$2 \times ((n-1) + (n-2) + (n-3) + ... + (n-n));$$

by comparison, partitioning an interaction between two compositions of one level only reduces the number of interface exchanges in the best case to

$$2 + 2 \times ((n-2) + (n-4) + (n-6) + ... + (n-n)).$$

This is more clearly indicated in Fig. 11 which illustrates the number of interface exchanges for interactions involving from 2 to 6 SMCs, arranged either as peer-to-peer collaborations or compositions.

This indicates that the use of architectural patterns mitigates the problems of scaling to larger systems, with respect to both programming complexity and the number of interactions that must be established among components. Engineering SMC interactions through the use of patterns thus provides a measurable gain over unstructured solutions.

5.3 Memory Consumption and Performance

To evaluate the performance of our prototype in resources with limited computational power and memory, we used two classes of lightweight, constrained devices:

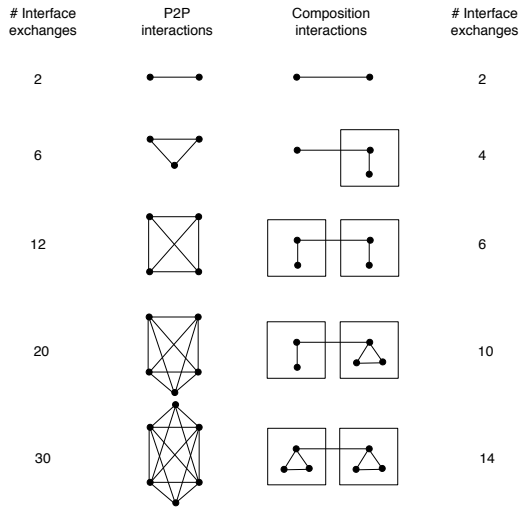


Fig. 11 Interface exchanges in compositions and peer-to-peer interactions

Gumstix[5] and Koala robots[6]. The Gumstix has a 400 MHz Intel XScale PXA255 processor with 16 MB flash memory and 64 MB SDRAM, running Linux and Wi-Fi enabled. The Koala robot has a Motorola 68331, 22 MHz onboard processor, 1 MB ROM and 1 MB RAM. The robot is extended with a KoreBot module which has a 400 MHz ARM PXA255 processor, 64 MB SDRAM and 32 MB flash memory, running Linux and also Wi-Fi enabled. In addition, the robot has 16 infrared proximity sensors around its body, and a video camera. Both run the lightweight JamVM[7]. In our setting we used JamVM version 1.4.5 and GNU Classpath version 0.91.

The size of the bytecodes required for running the prototype, including Ponder2 and necessary Java libraries, is 710 KB. The size of a typical policy written in Ponder2 is about 620 bytes (but this certainly depends on the complexity of the policy). The size of a typical interaction specification containing 5 roles, each role specifying 5 policies, written in Ponder2 is about 20.4 KB (but this is also subject to the complexity of the policies, number of policies, and number of roles in the specification). In terms of memory usage during run-time, we observed that a Gumstix keeping an interaction specification and all the objects loaded in memory, required 15 MB for the Ponder2 process and 9,224 KB for the *rmiregistry* process[8] (*RMI* is one of the communication protocols supported by Ponder2, and the one used in our experiments). A Koala robot running an application role (containing 5 policies) required 8,384 KB for the Ponder2 process and 4,492 KB for the *rmiregistry* process. Increasing the number of policies loaded in the robot from 5 to 10 caused a negligible overhead in terms of memory consumption. The

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

[6] http://www.k-team.com

[7] http://jamvm.sourceforge.net

[8] By comparison, an empty JamVM and rmiregistry uses about 3,200 KB and 5,900 KB respectively, and a JamVM running an empty Ponder2 instance and rmiregistry uses about 8,200 KB and 5,900 KB respectively.

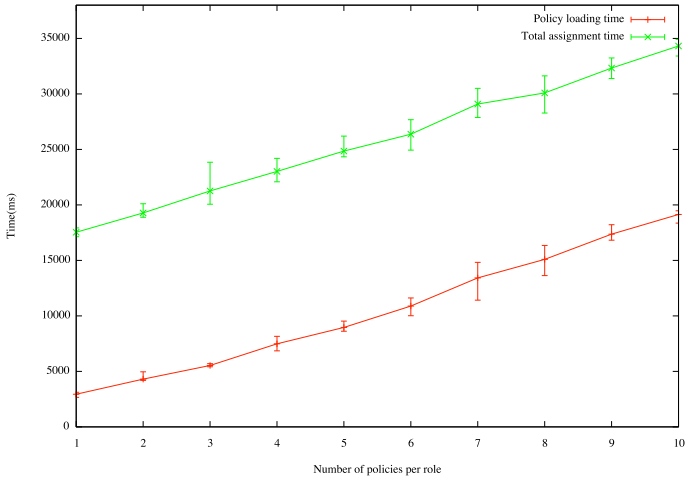


Fig. 12 Total assignment time *versus* policy downloading and deployment time

small footprint needed for our role management infrastructure indicates that other devices with a similar configuration and capacity could also have been used.

Performance tests were executed to measure the time taken for a Gumstix to assign a discovered Koala robot to a role, and then to load a variable number of policies. The graph in Fig. 12 depicts our results. We have measured both the time taken to transfer and deploy only the policies, as well as the whole assignment process. The latter involves the transfer of the policies, the transfer of additional information such as event templates, the creation of role placeholders in the remote SMC, sending an event informing that a new SMC has joined the interaction, and the attribution of the discovered SMC to the role in question.

Our results show that for roles with a small number of policies the total cost of assignment is dominated by the cost of tasks not related to policy transfer, i.e., the creation of role placeholders and the exchange of management information. However, as we increase the number of policies per role, this cost remains constant in comparison to the cost of transferring policies, which as can be observed in the graph increases linearly with the number of policies. Thus, for roles with a higher number of policies, the costs related to the creation of role placeholders and the exchange of management information tend to become less important. This suggests that the prototype is able to support more complex roles where the only significant cost is the policy transfer, because the residual component of the assignment time remains constant. We also observed that most of this time (about 97% on average) is spent on RMI serialization and network delay when transferring data from the Gumstix to the robot, and only a small part corresponds to the time that is actually spent by the robot to instantiate the policies. We expect that Ponder2's ability of supporting alternative communication protocols will mitigate this overhead. The evaluation of other aspects of the strategy, in particular the cost of role replacement when an SMC fails, remains to be done as future work.

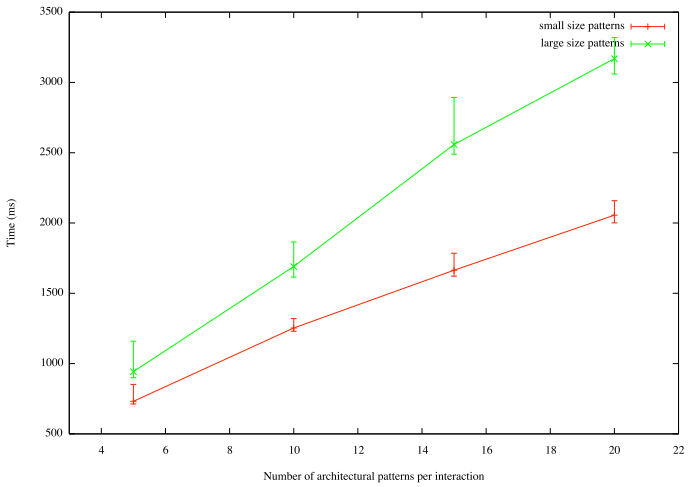


Fig. 13 Establishment time for deployments of increasingly large SMC interactions

5.4 Scalability of Large SMC Deployments

To further assess the scalability of the prototype we evaluated it with respect to the establishment time for deployments of increasingly large SMC interactions. The experiments were performed on a MacBook Pro 2.4GHz Intel Core 2 Duo with 8GB RAM memory. In particular, we use five SMCs to execute the interactions in these experiments, as this could typically represent, for example, the interactions between doctor and patient devices, with a small set of sensors and actuators. Each SMC runs on its own Java VM, however in these experiments all SMCs are executing on the same host.

In terms of workload, we characterise the scale of SMC interactions based on two factors: the *number of architectural patterns* used to combine the SMCs (`5`, `10`, `15`, `20`), and the *size of each pattern* (`small` and `large`). More specifically, we determine the size of an architectural pattern based on the number of exchanges of interface elements/events/policies performed by that specific pattern. In our experiments, a small-size pattern performs 4 exchanges, whereas a large pattern performs 8 exchanges.

We measured the interaction establishment time for each scenario, varying the number of architectural patterns, both using small and large patterns. Each experiment was run 30 times, amounting to a total of 240 executions. Fig. 13 shows the mean establishment time for different numbers and sizes of patterns (vertical lines represent minimum and maximum values for each experiment), and Table 2 summarises the mean establishment time, standard deviation and 95% confidence interval for deployments of increasingly large SMC interactions. For example, an interaction performing the establishment of 5 patterns, where each pattern is based on the exchange of four policies, events or interface elements, takes on average 731.93 milliseconds to complete. As we increase the number of patterns (but still maintaining four exchanges per pattern), we observe that the mean establishment time grows linearly and an interaction involving 20 patterns takes

2,055.73 milliseconds to complete. The behaviour of interactions based on larger patterns (involving the exchange of eight policies, events or interface elements) also appears to grow linearly as we increase the number of patterns, however the growth is steeper in this case, as can be observed in Fig. 13.

Table 2 Mean establishment time, standard deviation and 95% confidence interval for deployments of SMC interactions

Small size patterns	5 patterns	10 patterns	15 patterns	20 patterns
Mean (ms)	731.93	1253.23	1664.13	2055.73
Std dev	27.75	21.08	30.62	42.80
95% CI	[721.59;742.28]	[1245.37;1261.09]	[1652.72;1675.55]	[2039.78;2071.69]
Large size patterns	5 patterns	10 patterns	15 patterns	20 patterns
Mean (ms)	942.67	1691.10	2558.77	3169.53
Std dev	59.3501	46.5939	85.8209	57.0872
95% CI	[920.54;964.79]	[1673.73;1708.47]	[2526.77;2590.76]	[3148.25;3190.82]

Our results suggest that the prototype is able to support more complex interactions, in which the interaction establishment time increases linearly as a function of the number of patterns and the number of exchanges performed by each pattern. The results also show that the overhead for performing policy and interface exchanges and for setting up the communication for event exchanges remains within acceptable bounds (i.e., an interaction that exchanges 8 policies, 2 sets of interface exchanges with four elements each, and sets up the communication of 24 events – 10 small-size patterns – takes about 1.25 seconds to complete).

6 Related Work

The work described in this paper embraces several research areas, including engineering and design of ubiquitous services, multi-agent systems and software architecture-based approaches. These are discussed in the following.

In [35], the authors propose a new network architecture for deploying ubiquitous services by designing a modular and extensible framework of roles. By means of plug-in new roles [36], the architecture can evolve to fulfil new requirements, dynamically adding new features and modes of operation. In contrast to our work, those roles are typically associated to network layers 1-7 services, instead of focusing on policy-based interactions. Also, in [35] there is only minimum ordering rules amongst certain dependent roles, whereas we specifically focus on collaborating roles in the context of an interaction. The notions of Netlets [37] and autonomic functional blocks (AFBs) [38] aim to support new architectures for ubiquitous systems and for the Future Internet, and are both inspired by component-based software development approaches. They promote the composition of protocols out of so-called building blocks during design-time. Components are collected within a design repository for further reuse. However, both Netlets and AFBs components are aimed at the specification of network stack protocol functionality.

Work on multi-agents has investigated the use of organisational structures for designing multi-agent systems. Holonic models [39] for example often support

hierarchical structures, but not more sophisticated abstractions. Few studies however have attempted to identify a catalogue of generic and reusable patterns for agent interactions [40]. In [41], the authors present a preliminary catalogue of patterns for collaborations in a multi-agent system. Those patterns are presented in terms of how the goals of a collaboration are specified, and include, for example, "swarm intelligence" (goals are implicit) and "negotiation" (goals are explicit). The Gaia methodology [42], for example, supports the modelling and the development of multi-agent systems that could undergo organisational adaptations – an organisation describes the control regime of agent interactions, e.g., command-based, peer-based, market-based, norm-based, etc. Whereas [42] is concerned with general-purpose agent interactions, our focus is on policy-based interactions. The SMC resembles a *sentient object* [43] in that both are intended to model a set of interacting hardware and software components, and provide an infrastructure to support large-scale distributed systems composed of mobile autonomous components. However, while SMCs can dynamically exchange policies to define how remote SMCs must interact, sentient objects only rely on static rules. Also, sentient objects follow a *WAN-of-CANs* [44] structure (wide-area network of local controller-area networks) and cannot be dynamically assembled using more general and reusable patterns of interaction. Furthermore, in the system presented in [45], the notion of location context is used to support a location-based publish/subscribe (LPS) mechanism to address geographic dependency between distributed components. Natively, the SMC model does not offer support for delivery of events based on proximity, but this could be implemented in our framework as an LPS communication pattern.

To support the systematic construction of policy-based SMC interactions we also sought inspiration from the software engineering area. This community has long investigated software architecture-based approaches, which separate computation (*components*) from interactions (*connectors*). The benefits brought by this distinction have been widely recognised as a means of structuring software development [22]. Intuitively, interaction patterns could be seen as the hierarchical composition of distributed programmable connectors. The work presented in [46] also uses Alloy for model-checking the architectural properties of a software system, including pattern consistency, validity of specific structures, equivalence of global and local constraints, and checking for compatibility between patterns. The authors of [46] developed a toolset that automatically maps an architectural pattern specification to an Alloy model; at the moment, patterns in the SMC model have to be manually ported to Alloy for verification. However, components and connectors are low level implementation abstractions for general-purpose software interactions. In contrast, we have focussed on the compositionality of management and adaptation interactions. Some recent efforts have proposed a software architecture-based approach for engineering heterogeneous robotics systems [47]. They rely on architectural abstractions such as peer-to-peer and client-server to enable the exchange of high-level design solutions and assist in the creation and reuse of hierarchical components. None of these efforts, however, addresses patterns for building policy-based interactions in specific.

Finally, although the benefits of the use of architectural patterns for building policy-based systems are irrefutable, patterns are certainly not the only software engineering principle that is important for good design practices and that we have sought to apply and promote. The *dependency inversion principle* [48]

states that high-level modules should not depend on low-level modules, but instead they should both depend on abstractions. Dependency inversion aims to minimise system-wide changes due to cascading effects when a component is modified, reduce the possibility of breaking one part of the system due to changes in other parts, and facilitate reuse of components. In the SMC framework, dependency inversion is used for example in the specification of missions, which are high-level specifications that are decoupled from the low-level SMCs, via the *role abstraction*. *Liskov substitution principle (LSP)* [49] intends to guarantee semantic interoperability of types in a hierarchy. This principle defines that objects of a type T may be replaced with objects of type S, provided S is a subtype of T. Liskov substitution is related to the *open/closed principle* [50], which states that well-designed code can be extended by adding new code, rather than by changing already working code. If a module does not conform to LSP, then that module uses a reference to a base type but must know about all the subtypes of that type. This in turn violates the open/closed principle in the sense that the module would have to be modified whenever a new subtype is created. Thus ensuring that a *role hierarchy* satisfies LSP is a desirable property, because the application programmer can define policies and missions that will work for any subtype of a role. Another important design aspect necessary for building large-scale systems is the *interface-segregation principle* [48]. It defines that a component should not be forced to depend on methods it does not use. Interface-segregation is used in the SMC model to split large SMC interfaces into more specific ones such that remote components will only have to know about the methods that are of interest to them. Also important for supporting the specification of large-scale systems and component reuse is the *single responsibility principle* [48], which determines that every component should have a single responsibility, and that responsibility should be entirely encapsulated by the component. The rationale behind this principle is that if a component has more than one responsibility, it might need to be modified due to more than one reason. Single responsibility thus implies that coupling in a single component two implementation aspects that might change independently for different reasons is a bad design choice. We advocate in this paper that the proper use of patterns and relevant software design abstractions can assist engineers to build more scalable, robust and reusable components of a policy-based management system.

7 Concluding Remarks

This paper presented an integrated framework for supporting the design and the rapid establishment of policy-based interactions between management components of autonomous systems. We distinguish between the overall organisation of the interaction (structural aspects), the manner in which policies are exchanged (task-allocation aspects) and how events are forwarded between SMCs (communication aspects). These can be seen as complementary perspectives of a policy-based interaction, and for each of them we propose the use of interaction patterns that can be independently specified, instantiated and reused to form larger SMC collaborations. Whilst we focus here on the interaction patterns and their combination to compose SMCs, other aspects such as coordination, orchestration and governance

of distributed autonomous systems require further investigation and remain in the domain of future work.

Although the identification and specification of patterns does require human involvement, this is unavoidable because frequently best design practices tend to be domain-specific and dependent on experience. In [51,52], the authors proposed a system that reorganises network configurations into a more manageable configuration. It aims to identify and group common policies by discovering a set of shared features between them. In principle, a similar approach could be used to assist in the early identification of policy interactions in an existing system, which could be subsequently encoded as reusable patterns.

Devolved management is the key for addressing the complexity of large-scale networked systems which are formed as collaborations of smaller, yet autonomous, components. This paper investigated how policy-based autonomous components can be federated and composed to form larger applications. This relies on previous research to address a new problem: *engineering policy-based autonomous systems*. We adapted and incorporated techniques from autonomous systems, multi-agents and software engineering principles, and identified how these studies could benefit the construction of policy-based systems. The use of patterns for systematically building policy-based systems is a novel and promising approach. Although we have concentrated on management of *Self-Managed Cells*, the principles and techniques proposed provide initial insights towards the engineering process of ubiquitous and autonomous systems in general. Note however that a more general theory of the composition of self-managed software systems is beyond the scope of this paper. Research communities such as the SEAMS symposium [53] aim to bring together researchers working towards this more fundamental goal.

More recently, we have shown how the architectural abstractions presented in this paper can be applied to compose policy-based autonomous systems for network security and resilience, and in particular to address network-wide anomalies such as Distributed Denial of Service (DDoS) attacks and worm propagations [54, 55]. Our focus on these aspects is in part motivated by the recurrence of common issues and patterns across the application and deployment of security techniques [56–58]. Further work on the use of management patterns for enforcing reusable configurations for security and resilience properties is ongoing.

Acknowledgements This work was partly funded by the UK Engineering and Physical Sciences Research Council through grant GR/S68040/01; the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001; and the EC IST EMANICS Network of Excellence (#26854).

References

1. E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, A. Schaeffer-Filho, AMUSE: autonomic management of ubiquitous systems for e-health, Concurrency and Computation: Practice and Experience, John Wiley 20(3) (2008) 277–295.
2. IBM, An architectural blueprint for autonomic computing. third edition, Tech. rep., IBM (June 2005).
3. M. Sloman, E. Lupu, Engineering policy-based ubiquitous systems, The Computer Journal 53 (7) (2010) 1113–1127. doi:10.1093/comjnl/bxp102.

4. E. Asmare, A. Gopalan, M. Sloman, N. Dulay, E. Lupu, Self-management framework for mobile autonomous systems, Journal of Network and Systems Management 20 (2012) 244–275. doi:10.1007/s10922-011-9201-5.
URL http://dx.doi.org/10.1007/s10922-011-9201-5
5. A. Schaeffer-Filho, E. Lupu, N. Dulay, S.-L. Keoh, K. Twidle, M. Sloman, S. Heeps, S. Strowes, J. Sventek, Towards supporting interactions between self-managed cells, in: Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE Computer Society, Boston, USA, 2007, pp. 224–233.
6. A. Schaeffer-Filho, E. Lupu, M. Sloman, Realising management and composition of self-managed cells in pervasive healthcare, in: Pervasive Computing Technologies for Healthcare, 2009. PervasiveHealth 2009. 3rd International Conference on, 2009, pp. 1 –8. doi:10.4108/ICST.PERVASIVEHEALTH2009.5979.
7. A. Schaeffer-Filho, E. Lupu, M. Sloman, S. Eisenbach, Verification of policy-based self-managed cell interactions using alloy, in: Proceedings of the 10th IEEE international conference on Policies for distributed systems and networks, POLICY'09, IEEE Press, Piscataway, NJ, USA, 2009, pp. 37–40.
URL http://dl.acm.org/citation.cfm?id=1812664.1812673
8. J. Ma, A. Russo, K. Broda, K. Clark, Dare: a system for distributed abductive reasoning, Autonomous Agents and Multi-Agent Systems 16 (3) (2008) 271–297. doi:10.1007/s10458-008-9028-y.
URL http://dx.doi.org/10.1007/s10458-008-9028-y
9. C. Hwang, E. Talipov, H. Cha, Distributed geographic service discovery for mobile sensor networks, Computer Networks 55 (5) (2011) 1069 – 1082. doi:10.1016/j.comnet.2010.09.015.
10. C. Esposito, D. Cotroneo, S. Russo, On reliability in publish/subscribe services, Computer Networks (0) (2013) –. doi:10.1016/j.comnet.2012.10.023.
11. M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The enterprise service bus: making service-oriented architecture real," *IBM Syst. J.*, vol. 44, no. 4, pp. 781–797, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1147/sj.444.0781
12. J. Singh, D. M. Eyers, J. Bacon, Disclosure control in multi-domain publish/subscribe systems, in: Proceedings of the 5th ACM international conference on Distributed event-based system, DEBS '11, ACM, New York, NY, USA, 2011, pp. 159–170. doi:10.1145/2002259.2002283.
URL http://doi.acm.org/10.1145/2002259.2002283
13. Y. Zhu, S. L. Keoh, M. Sloman, E. Lupu, A lightweight policy system for body sensor networks, IEEE Transactions on Network and Service Management 6 (3) (2009) 137 –148.
14. S.-L. Keoh, E. Lupu, M. Sloman, Securing body sensor networks: Sensor association and key management, in: Proceedings of the IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 1–6. doi:http://doi.ieeecomputersociety.org/10.1109/PERCOM.2009.4912756.
15. R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *7th International Conference on Network and Service Management (CNSM)*. IEEE, 2011, pp. 1–9.
16. D. Corapi, O. Ray, A. Russo, A. Bandara, and E. Lupu, "Learning rules from user behaviour," in *Artificial Intelligence Applications and Innovations III*, ser. IFIP International Federation for Information Processing, Iliadis, Maglogiann, Tsoumakasis, Vlahavas, and Bramer, Eds. Springer US, 2009, vol. 296, pp. 459–468.
17. J. Ma, K. Broda, A. Russo, and E. Lupu, "Distributed abductive reasoning with constraints," in *Declarative Agent Languages and Technologies VIII*, ser. Lecture Notes in Computer Science, A. Omicini, S. Sardina, and W. Vasconcelos, Eds. Springer Berlin Heidelberg, 2011, vol. 6619, pp. 148–166. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-20715-0_9
18. M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. Bandara, E. Lupu, A. Russo, N. Dulay, and M. Sloman, "Policy conflict analysis for diffserv quality of service management," *Network and Service Management, IEEE Transactions on*, vol. 6, no. 1, pp. 15–30, March 2009.
19. M. Stal, "Web services: beyond component-based computing," *Communications of the ACM*, vol. 45, no. 10, pp. 71–76, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/570907.570934
20. B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

21. E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Professional Computing Series, Addison-Wesley, 1995, 416 pages.
22. R. N. Taylor, N. Medvidovi, I. E. Dashofy, Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2009.
23. S. Salah, G. Macia-Fernandez, J. E. Diaz-Verdejo, A model-based survey of alert correlation techniques, Computer Networks (0) (2013) –. doi:10.1016/j.comnet.2012.10.022.
24. M. Cinque, C. D. Martino, C. Esposito, On data dissemination for large-scale complex critical infrastructures, Computer Networks 56 (4) (2012) 1215 – 1235. doi:10.1016/j.comnet.2011.11.016.
25. A. Schaeffer-Filho, Supporting management lnteraction and composition of self-managed cells, Ph.D. thesis, Imperial College London, London, UK (2009).
26. D. Jackson, Alloy: a lightweight object modelling notation, ACM Transactions on Software Engineering Methodologies 11 (2) (2002) 256–290. doi:http://doi.acm.org/10.1145/505145.505149.
27. D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.
28. R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I, Information and Computation 100 (1) (1992) 1–40. doi:http://dx.doi.org/10.1016/0890-5401(92)90008-4.
29. L. Cardelli, A. D. Gordon, Mobile ambients, in: Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structure (FoSSaCS), Springer-Verlag, London, UK, 1998, pp. 140–155.
30. A. Phillips, Specifying and implementing secure mobile applications in the channel ambient system, Ph.D. thesis, Imperial College London (April 2006).
31. J. M. Spivey, *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.
32. D. Bjørner and C. B. Jones, Eds., *The Vienna Development Method: The Meta-Language*. London, UK, UK: Springer-Verlag, 1978.
33. K. Lano, *The B Language and Method: A Guide to Practical Formal Development*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.
34. Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs, ACM Transactions on Software Engineering Methodologies 11 (2) (2002) 215–255. doi:http://doi.acm.org/10.1145/505145.505148.
35. X. Sanchez-Loro, J. L. Ferrer, C. Gomez, J. Casademont, J. Paradells, Can future internet be based on constrained networks design principles?, Computer Networks 55 (4) (2011) 893 – 909, special Issue on Architectures and Protocols for the Future Internet. doi:10.1016/j.comnet.2010.12.018.
36. X. Sanchez-Loro, A. Gonzalez, R. Martin-De-Pozuelo, A semantic context-aware network architecture, in: Future Network and Mobile Summit, 2010, 2010, pp. 1 –9.
37. D. Martin, L. Volker, M. Zitterbart, A flexible framework for future internet design, assessment, and operation, Computer Networks 55 (4) (2011) 910 – 918, special Issue on Architectures and Protocols for the Future Internet. doi:10.1016/j.comnet.2010.12.015.
38. M. Sifalakis, A. Louca, G. Bouabene, M. Fry, A. Mauthe, D. Hutchison, Functional composition in future networks, Computer Networks 55 (4) (2011) 987 – 998, special Issue on Architectures and Protocols for the Future Internet. doi:10.1016/j.comnet.2010.12.006.
39. V. Hilaire, A. Koukam, S. Rodriguez, An adaptative agent architecture for holonic multiagent systems, ACM Transactions on Autonomous and Adaptive Systems 3 (1) (2008) 1–24. doi:http://doi.acm.org/10.1145/1342171.1342173.
40. F. Zambonelli, N. R. Jennings, M. Wooldridge, Developing multiagent systems: The Gaia methodology, ACM Transactions on Software Engineering Methodologies 12 (3) (2003) 317–370. doi:http://doi.acm.org/10.1145/958961.958963.
41. G. Cabri, M. Puviani, F. Zambonelli, Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles, in: Collaboration Technologies and Systems (CTS), 2011 International Conference on, 2011, pp. 508 –515. doi:10.1109/CTS.2011.5928730.
42. L. Cernuzzi, A. Molesini, A. Omicini, F. Zambonelli, Adaptable multi-agent systems: The case of the gaia methodology, International Journal of Software Engineering and Knowledge Engineering 21 (04) (2011) 491–521. doi:10.1142/S0218194011005384.
43. A. F. Gregory, G. Biegel, S. Clarke, V. Cahill, Towards a sentient object model, in: In Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE'2002), 2002.

44. P. Veríssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, J. Kaiser, Cortex: Towards supporting autonomous and cooperating sentient entities, in: Proceedings of European Wireless 2002, Florence, Italy, 2002, pp. 595–601.
45. A. Holzer, P. Eugster, B. Garbinato, Alps – adaptive location-based publish/subscribe, Computer Networks 56 (12) (2012) 2949 – 2962. doi:10.1016/j.comnet.2012.05.007.
46. J. S. Kim, D. Garlan, Analyzing architectural styles, Journal of Systems and Software 83 (7) (2010) 1216–1235. doi:10.1016/j.jss.2010.01.049.
47. N. Medvidovic, H. Tajalli, J. Garcia, I. Krka, Y. Brun, G. Edwards, Engineering heterogeneous robotics systems: A software architecture-based approach, Computer 44 (5) (2011) 62–71. doi:10.1109/MC.2010.368.
URL http://dx.doi.org/10.1109/MC.2010.368
48. R. C. Martin, Agile Software Development: Principles, Patterns, and Practices. Prentice-Hall (2002).
49. B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Trans Program Lang Syst 16 (6) (1994) 1811–1841. doi:10.1145/197320.197383.
50. R. C. Martin, The open-closed principle, Cambridge University Press (2000), New York, NY, USA, pp 97–112. URL http://dl.acm.org/citation.cfm?id=331120.331143
51. S. Lee, T. Wong, H. S. Kim, Improving manageability through reorganization of routing-policy configurations, Computer Networks 56 (14) (2012) 3192 – 3205. doi:10.1016/j.comnet.2012.06.014.
52. S. Lee, T. Wong, H. Kim, Improving dependability of network configuration through policy classification, in: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, 2008, pp. 297 –306. doi:10.1109/DSN.2008.4630098.
53. *SEAMS '13: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Piscataway, NJ, USA: IEEE Press, 2013.
54. A. Schaeffer-Filho, P. Smith, A. Mauthe, D. Hutchison, Y. Yu, M. Fry, A framework for the design and evaluation of network resilience management, in: Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium (NOMS 2012), IEEE Computer Society, Maui, Hawaii, USA, 2012, pp. 401–408.
55. A. Schaeffer-Filho, P. Smith, A. Mauthe, and D. Hutchison, "Network resilience with reusable management patterns (to appear)," *IEEE Communications Magazine*, July 2014.
56. Y. Zhou, Y. Fang, Y. Zhang, Securing wireless sensor networks: a survey, IEEE Communications Surveys & Tutorials 10 (3) (2008) 6 –28. doi:10.1109/COMST.2008.4625802.
57. J. P. Sterbenz, D. Hutchison, E. K. Cetinkaya, A. Jabbar, J. P. Rohrer, M. Scholler, P. Smith, Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines, Computer Networks 54 (8) (2010) 1245 – 1265, resilient and Survivable networks. doi:10.1016/j.comnet.2010.03.005.
58. X. Chen, K. Makki, K. Yen, N. Pissinou, Sensor network security: a survey, IEEE Communications Surveys & Tutorials 11 (2) (2009) 52–73. doi:10.1109/SURV.2009.090205.
URL http://dx.doi.org/10.1109/SURV.2009.090205

Author Biographies

Alberto Schaeffer Filho is an Associate Professor in the Institute of Informatics at Federal University of Rio Grande do Sul (UFRGS). Prior to that he was a Research Associate in Lancaster University for three years. He obtained his PhD in Computing from Imperial College London in 2009. His research interests include network management, autonomous systems, software engineering principles, security and resilience of networks. He is a member of the IEEE. See http://www.inf.ufrgs.br/∼alberto for more details and selected papers.

Emil Lupu is a Reader in Adaptive Computing Systems and Associate Director of the Institute for Security Science and Technology at Imperial College London. He leads several research projects in the areas of pervasive computing, adaptive and autonomous systems, trust and security. He has over 120 publications in these areas, serves on the editorial boards of the IEEE Trans. on Network and Service Management, Journal of Network and Systems Management

and the International Journal of Network Management, and on the program committees of numerous conferences. See http://www.imperial.ac.uk/people/e.c.lupu for further details and selected publications.

Morris Sloman is a Professor of Distributed Systems Management, and Deputy Head of the Department of Computing, Imperial College London. His research interests include autonomic management of pervasive systems, adaptive security management, privacy and security for pervasive systems. He is a member of the editorial board of the Journal of Network and Systems Management and IEEE Transactions on Network and Services Management. See http://www.doc.ic.ac.uk/∼mss for selected papers.