University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Supporting Management Interaction and Composition of Self-Managed Cells

Alberto Egon Schaeffer Filho

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, September 2009

Abstract

Management in ubiquitous systems cannot rely on human intervention or centralised decision-making functions because systems are complex and devices are inherently mobile and cannot refer to centralised management applications for reconfiguration and adaptation directives. Management must be devolved, based on local decision-making and feedback control-loops embedded in autonomous components. Previous work has introduced a *Self-Managed Cell (SMC)* as an infrastructure for building ubiquitous applications. An SMC consists of a set of hardware and software components that implement a policy-driven feedback control-loop. This allows SMCs to adapt continually to changes in their environment or in their usage requirements. Typical applications include *body-area networks* for healthcare monitoring, and communities of *unmanned autonomous vehicles (UAVs)* for surveillance and reconnaissance operations.

Ubiquitous applications are typically formed from multiple interacting autonomous components, which establish peer-to-peer collaborations, federate and compose into larger structures. Components must interact to distribute management tasks and to enforce communication strategies. This thesis presents an integrated framework which supports the design and the rapid establishment of policy-based SMC interactions by systematically composing simpler abstractions as building elements of a more complex collaboration. Policy-based interactions are realised – subject to an extensible set of *security functions* – through the exchanges of interfaces, policies and events, and our framework was designed to support the specification, instantiation and reuse of *patterns of interaction* that prescribe the manner in which these exchanges are achieved. We have defined a library of patterns that provide reusable abstractions for the structure, task-allocation and communication aspects of an interaction, which can be individually combined for building larger policy-based systems in a methodical manner. We have specified a *formal model* to ensure the rigorous verification of SMC interactions before policies are deployed in physical devices. A prototype has been implemented that demonstrates the practical feasibility of our framework in constrained resources.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Emil Lupu. This thesis would not have been possible without his support, advice and guidance. His constructive criticism and dedication were inspiring and motivational. I am also very thankful to my second supervisor, Professor Morris Sloman, whose advice and invaluable insights helped me to improve my understanding of my research area. I would like to express my wholehearted gratitude to both Emil and Morris for organising the financial support required for me to pursue my PhD studies.

I am grateful to the academic staff and researchers in our department for their guidance and technical advice, in particular Dr. Naranker Dulay, Professor Susan Eisenbach, Dr. Alessandra Russo and Dr. Kevin Twidle.

Many thanks to my fellow students and friends in the Distributed Software Engineering (DSE) section, namely Daniel Sykes, Leonardo Mostarda, Eskindir Asmare, Sye-Loong Keoh, Paulo Henrique Maia, Anandha Gopalan, Enrico Scalavino, Robert Craven, Srdjan Marinovic, Lucio Mauro Duarte, Markus Huebscher, Vrizlynn Thing, Changyu Dong, Howard Foster, Domenico Corapi, Rudi Ball, Jiefei Ma and William Heaven for their friendship and support. Thanks to all members of the DSE section for making this a wonderful experience. Special thanks to Daniel and Robert for proof-reading chapters of this thesis.

I am deeply thankful to my family for their constant care, encouragement and for understanding my reasons for being away for so long. My most profound gratitude goes to my parents, Alberto Schaeffer (in memory) and Adelia Debarba Juchem, to whom this thesis is dedicated. They provided me with the inspiration to never give up. Finally, I would like to thank Amanda Superti for being always supportive and for sharing with me most moments of my life outside Imperial.

The work presented in this thesis 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).

Statement of Contribution

The contents of this thesis are the result of the author's participation in the work developed at Imperial College on policy-based management in ubiquitous systems over the last four years. This is based on the *Self-Managed Cell* (SMC), which was first proposed by Dr. Emil Lupu, Professor Morris Sloman and Dr. Naranker Dulay, in conjunction with Professor Joe Sventek from the University of Glasgow, in the context of the EPSRC project *AMUSE: Autonomic Management of Ubiquitous Systems for e-Health*.

This thesis is motivated by the need to support the design, instantiation and reuse of policy-based SMC interactions, thereby allowing SMCs to federate and compose into larger structures. This work builds upon the basic SMC architecture and its core services, which are described in Chapter 2 and based on previous work which is credited accordingly. This thesis identifies and develops the underlying mechanisms for supporting SMC interactions, which did not exist in the basic SMC architecture. The use of patterns for systematically defining SMC interactions described in Chapter 4 and the formal specification of the SMC behaviour presented in Chapter 5 are also the author's individual work.

The implementation of the framework presented in this thesis uses the Ponder2 policy framework, developed by Kevin Twidle, as the underlying platform for supporting SMC interactions. Implementations of the SMC core services, in particular the discovery service developed by Sye-Loong Keoh and the event bus developed by Stephen Strowes, were also used. The Ponder2 extensions required for implementing the principles and techniques proposed in this thesis are the author's individual work.

List of Publications

Book Chapters

- A. Schaeffer-Filho, E. Lupu, and M. Sloman. "Network Science for Military Coalition Operations: Information Extraction and Interactions", chapter *Policies to Enable Secure Dynamic Community Establishment* (to appear).

Conferences and Workshops

- A. Schaeffer-Filho, E. Lupu, M. Sloman, and S. Eisenbach, "Verification of Policy-based Self-Managed Cell Interactions Using Alloy", in *Proceedings of the 10th IEEE International Symposium on Policies for Distributed Systems and Networks (Policy 2009).* London, UK, July 2009.

- A. Schaeffer-Filho, E. Lupu, and M. Sloman, "Realising Management and Composition of Self-Managed Cells in Pervasive Healthcare", in *Proceedings of the 3rd International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth 2009).* London, UK. IEEE, April 2009.

- A. Schaeffer-Filho, E. Lupu, M. Sloman, S.-L. Keoh, J. Lobo, and S. Calo, "A Role-based Infrastructure for the Management of Dynamic Communities", in *Proceedings of the 2nd International Conference on Autonomous Infrastructure, Management and Security (AIMS 2008)*, ser. LNCS. Bremen, Germany. Springer, July 2008, pp. 1–14.

- A. Schaeffer-Filho, E. Lupu, N. Dulay, S. L. Keoh, K. Twidle, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Towards Supporting Interactions between Self-Managed Cells", in *Proceedings of the 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, Boston, USA, July 2007, pp. 224–233.

- A. Schaeffer-Filho and E. Lupu, "Abstractions to Support Interactions between Self-Managed Cells", in *Proceedings of the 1st International Conference on Autonomous Infrastructure, Management and Security (AIMS 2007), doctoral symposium*, ser. LNCS. Oslo, Norway. Springer, June 2007, pp. 160–163.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ubiquitous systems typically comprise numerous portable devices, such as smartphones, sensors and electronic devices in general, which are inherently mobile and cannot refer to centralised management applications for their reconfiguration and decision-making functions. In addition, the configuration and management of these devices is too complex and cumbersome to be left to general computer users. A typical example is a *body-area network* for health-care monitoring, comprising sensors and actuators that must integrate with external monitoring and diagnostic devices. To manage the intricacies and the scale of these systems, they must be built from autonomous elements with devolved management responsibilities. This goes against the tradition of large-scale network management services, which are functionally integrated in centralised network operation centres by human administrators.

Work on *pervasive and ubiquitous systems* [RHC$^+$02, GSSS02] often revolves around centralised middleware services that offer supporting functionality for applications, but these studies generally ignore how applications can be built as dynamic ad-hoc collaborations of smaller and autonomous components. The management of such systems frequently relies on centralised services for the coordination of the resources available in the environment, which in turn must be appropriately configured by the user. In order to mitigate the problems related to centralised and manual management, *autonomic computing* investigates how a system can manage itself requiring minimal, if any, human intervention; this was termed *self-management*. Self-management aims to enable the system to configure, optimise, protect and heal itself autonomously [KC03]. This kind of devolved management is a promising approach to tackle the complexity of large-scale ubiquitous systems, where autonomous components must be able to perform local decision-making and automatically negotiate the required interactions with other components in their surroundings.

Previous work has introduced the *Self-Managed Cell* (SMC) [LDS+08] as an infrastructure for building ubiquitous computing applications. An SMC consists of a set of hardware and software components that are able to work autonomously, based on a *policy-driven feedback control-loop*. Policies are used to specify both adaptation rules that determine which management or reconfiguration actions must be performed in response to changes in the context, and access control rules that determine which actions can or cannot be performed on the resources that an SMC provides. Policies separate the management strategy from the implementation of the management actions, and new policies can be dynamically loaded into an SMC, which permits the modification of the run-time adaptation strategy without interrupting the SMC's operation.

Although SMCs are autonomous, applications typically require a large number of SMCs to collaborate, which must be able to interact with each other in complex ways, to federate or compose into larger structures. Examples include body-area networks, which typically consist of various devices such as complex physiological sensors that may be SMCs in their own right, smartphones, diagnostic devices and equipment available locally. Cross-SMC interactions permit the realisation of pervasive environments in which SMCs can engage in ad-hoc peer-to-peer collaborations with other SMCs, and can aggregate into larger autonomous structures, scaling SMC management to larger environments. The Self-Managed Cell defines an architectural pattern for building policy-based autonomous systems composed of services interacting over an asynchronous event bus. An SMC can be instantiated to individual devices, as well as scale up to cater for management in larger ubiquitous systems.

To support applications that demand the cooperation of a number of elementary autonomous components it is thus necessary to devise abstractions to facilitate interactions between SMCs, allowing SMCs to compose and federate into large-scale policy-based systems which are also SMCs. Policies provide a method for adapting the SMC's behaviour according to changes of state in its local context, and also define how SMCs should behave when they dynamically encounter new SMCs. In these interactions, SMCs *exchange policies* to prescribe how a remote SMC must behave; SMCs can *forward events* which are required for notifying changes of context and triggering management policies in another SMC, and SMCs *provide interfaces* which offer functionality or facilitate access to the resources that the SMC owns. In this thesis we present the infrastructure that makes SMC interactions possible. To address the design and the rapid establishment of complex policy-based SMC interactions we advocate ways of structuring these collaborations, and the specification, instantiation and reuse of *patterns of interaction*, which prescribe the algorithms or protocols through which the exchanges of policies, events and interfaces are achieved. The use of patterns supports the construction of more complex SMC interactions in a methodical manner, by reusing simpler abstractions as design elements of a more complex interaction.

1.1 Example Applications

The application scenarios below will be used throughout this thesis to illustrate specific concepts relevant to collaborations of Self-Managed Cells. They describe how SMCs can be used in applications for *healthcare monitoring* and in *coalitions of unmanned autonomous vehicles*. While the former illustrates ways of organising and composing simpler SMCs into larger structures, e.g. body-area networks of sensors and actuators, the latter brings to light issues related to ad-hoc collaborations of mobile nodes and highlights the security aspects that must be considered in these interactions.

1.1.1 Healthcare Monitoring

Healthcare applications can be deployed both for automated monitoring in the hospital and for home monitoring of the elderly or patients with chronic conditions [LDS$^+$08]. A body-area SMC running on a smartphone can interact with a number of body-sensor SMCs monitoring properties such as temperature, heart rate, blood pressure, glucose and oxygen saturation. The body-area can also include actuator SMCs, such as a pacemaker or insulin pump, to perform actions in response to measurements made by the sensors. Local equipment available in the home environment, e.g. an ECG diagnostic device, may be used in conjunction with the body-area SMC to perform more complex data processing and condition assessment. A doctor or healthcare worker loads policy-based monitoring tasks into the patient, which rely on the devices available in the body-area SMC as well as on those in the home environment. Management policies are triggered in response to measurements made by the sensors, and then enforce actions for reporting abnormal physiological parameters, triggering a pacemaker or insulin infusion, or even automatically requesting emergency medical service if necessary. Similarly, management policies are used to reconfigure the SMC functioning in response to changes in its context, e.g. failures of components or performance degradation. The resources available in the body-area SMC can only be accessed by certain SMCs, such as the doctor or healthcare worker SMC, and in many cases they must be hidden from other SMCs nearby both for security and privacy reasons. Interactions would also be required when the patient SMC encounters an SMC controlling devices in the General Practitioner's (GP) clinic, which typically access the patient's resources to perform reconfiguration actions, e.g. to change dosage on drug delivery pumps.

1.1.2 Coalitions of UAVs

Coalitions of *unmanned autonomous vehicles (UAVs)* can be used in reconnaissance or rescue operations that are too dangerous or even impossible for humans to undertake [ADK+06, ADL+07]. An SMC running on each device enables local policy enforcement and decision-making in response to conditions monitored in the field, e.g. obstacles or chemical detection. Policies are used to govern the management actions taken by each individual vehicle, or to invoke actions on other UAVs in order to collaborate on the accomplishment of a mission. Policies can be directly loaded from a single commander UAV, or more elaborate task exchange mechanisms can be required, e.g. UAVs bidding for tasks according to their capabilities. Rescue personnel carry SMCs running on their portable communication devices, which interact with the UAVs and retrieve the information collected during the mission. Multiple teams of UAVs may need to collaborate to accomplish a mission, where some members participate in two different teams, e.g. in a UK and US coalition. This requires teams with relaxed restrictions on their structural organisation, allowing sharing of resources and equipment among cooperating teams. The nature of the coalition may demand more complex abstractions, for example a hierarchical organisation, in which a rescue team has as one of its members a medical team, which is a coalition itself.

1.2 Requirements Summary

We can draw several requirements from the scenarios described earlier. In order to form complex policy-based systems a number of autonomous SMCs must be able to federate with ***minimum or no user intervention***.

The SMCs in a federation must be able to ***exchange policies*** among themselves, as policies prescribe how an SMC should behave within the context of an interaction. Policies are triggered in response to events, which can be local events within the SMC or events generated by remote SMCs. Thus if management actions involved in the feedback control-loop are to span the boundaries of an SMC, federated SMCs must be able to ***exchange events*** among themselves and notify their partners of local context changes. The ***structure*** of these federations varies depending on the nature of the involved SMCs, where

SMCs may be required to share and expose some of their constituent resources to their partners, or at least mediate access to these resources in some cases.

Often a federation of SMCs will be part of a larger, and more complex interaction, such as a body-area network interacting with the equipment available at the GP, or two rescue teams of UAVs cooperating to accomplish a mission. Expressing large-scale interactions using simple abstractions such as policies, interfaces and events can be difficult to manage and deploy. Support is needed for the rapid establishment of complex policy-based SMC interactions, through the specification, instantiation and reuse of ***patterns of interaction*** that prescribe the manner in which these exchanges are achieved. Patterns can then be used for ***systematically composing abstractions*** as building elements of a more complex collaboration.

Some applications are more susceptible to security threats, e.g. coalitions of UAVs in a reconnaissance mission. These applications will necessitate stricter controls on which SMCs are allowed to join the interaction, and which are to be refused if they do not meet a set of criteria to ascertain their provenance. Collaborations of SMCs thus require support for ***secure interaction establishment and operation*** with distributed nodes enforcing devolved security functions.

Typically interactions will involve SMCs originating from different administrative authorities, and their successful operation depends both on the correct specification of their interactions and on the suitability of the participating SMCs. Ideally, interactions must be checked prior to implementation and policy deployment in physical devices. Ensuring the robustness of these interactions demands the ***verification*** of the interaction specification, and checking whether the collaborating SMCs are able to enforce their policies and whether these policies are correctly deployed among distributed SMCs.

Finally, an infrastructure is needed to enforce these interactions, i.e. an ***implementation***.

1.3 Contributions

This thesis makes the following contributions:

- The design of an integrated framework which supports the specification, instantiation and reuse of policy-based interactions by systematically composing simpler abstractions, using the Self-Managed Cell as the underlying infrastructure.
- The identification of the basic mechanisms required for supporting SMC interactions, in terms of dynamic loading of policies, the events required for triggering these policies, and the interfaces that offer the functionality required for enforcing the management actions prescribed by a policy.
- The definition of a catalogue of reusable patterns for engineering larger policy-based interactions that distinguishes 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).
- The definition of the minimum security aspects which must be considered in collaborations of SMCs, and how the framework can be extended to address additional requirements.
- The formal specification of the SMC behaviour, and the use of model-checking techniques to enable the verification of properties and check the correctness of SMC interactions before deployment.
- The implementation of a prototype and a library of interaction patterns that demonstrate how the principles and techniques presented in this thesis can be used for supporting the federation of SMCs.

Although this thesis focuses on the Self-Managed Cell framework as the infrastructure for building policy-based autonomous systems, the principles and techniques that are proposed here have a wider applicability for engineering pervasive and autonomous systems in general. This will become clear when the use of patterns is introduced, as patterns provide more general abstractions for structuring interactions between autonomous components.

1.4 Thesis Outline

This thesis is structured as follows. Chapter 2 presents related work, both on the Self-Managed Cell infrastructure and policy-based management in general, as well as on research areas that will be relevant at different stages of this thesis.

Chapter 3 investigates the underlying principles of SMC interactions. It describes the basic mechanisms that support interaction establishment and that allow an SMC to access the functionality provided by other SMCs. It describes how a group of policies can be loaded into an SMC to prescribe how it should behave in the context of an interaction, and how events are used to trigger these policies. The use of authorisation policies in SMC interactions is also presented, as policy loading and invocations into a remote SMC are subject to access control restrictions. This chapter concludes by illustrating how security management in SMC interactions can be achieved by using policies and roles to support secure interaction establishment and operation.

Chapter 4 introduces the use of software architecture principles for building large-scale policy-based interactions, using the notion of architectural styles. Styles can encode different aspects of an interaction, with respect to structure, task-allocation and event-forwarding behaviour. This chapter also describes how architectural styles can be used for composing and federating SMCs to form complex interactions in a systematic manner.

Chapter 5 presents the formalisation of SMC behaviour using the *Alloy* [Jac02] modelling language, which allows us to specify the functioning of SMC interactions declaratively. The specification can then be used for model checking specific SMC interactions and for verifying whether the SMCs in a collaboration are able to enforce their policies.

Chapter 6 provides details on the prototype implemented for supporting SMC interactions, and on the library of reusable patterns for composing SMCs. The chapter also presents the evaluation of the implementation, including memory consumption and performance results in resource-constrained devices.

Chapter 7 outlines a case-study presenting how SMCs can be used for building a policy-based application for diabetes monitoring and treatment. It shows how various SMCs can be composed in a body-area network, and the interactions which occur with a doctor or healthcare worker. The interactions between the body-area and a home monitoring SMC, involving a set of equipment and diagnostic devices, are also described.

Chapter 8 presents a more general discussion and critical evaluation of the framework proposed in this thesis, in terms of its usability, scalability and extensibility. The limitations and deficiencies of the framework are also discussed in this chapter.

Chapter 9 presents a short summary of this thesis, and how future work can make progress in some aspects that were not addressed yet. Finally, the closing remarks are presented.

Chapter 2

Background and Related Work

The Self-Managed Cell was proposed as a platform for the construction of policy-based autonomous systems. It is based on a policy-driven feedback control-loop that determines which management or reconfiguration actions must be performed in response to changes in the SMC context, e.g. failures of components or performance degradation. Individual SMCs are autonomous and charged with enforcing local decision rules that govern their own behaviour. However, applications will typically require elementary SMCs to negotiate the necessary interactions with other components in their surroundings and form larger systems based on the same principles of self-management.

This chapter presents the related work in a top-down approach: Section 2.1 is a general discussion of pervasive environments, their characteristics, and how autonomic computing aims to address the management aspects of these systems. Section 2.2 then describes policy-based management within this context, and Section 2.3 presents the background on Self-Managed Cells, which relies on policies to provide an infrastructure for autonomous management in pervasive and ubiquitous systems. Finally, Section 2.4 and Section 2.5 concentrate on specific techniques advocated for structuring the development of software, *component-based systems* and *multi-agent systems*. We identify how these research areas can benefit the systematic construction of policy-based SMC interactions. They provide the background for the engineering principles for composing SMCs in patterns of interactions, which are described in greater detail in Chapter 4.

2.1 Pervasive and Autonomous Systems

The work on Self-Managed Cells lies at the intersection of two broad research areas: *pervasive computing*, which seeks to provide smart environments saturated with technological capabilities, and *autonomic computing*, which investigates how to achieve the autonomous management of computing infrastructures, with minimal human intervention. This section presents a brief overview of these two research areas.

2.1.1 Pervasive Computing

Pervasive (or ubiquitous) computing concerns environments saturated with technological capabilities that are so integrated that they seem to *"disappear"* [Sat01]. A *"smart space"* is a well-defined area where the computational infrastructure is embedded in the building infrastructure, allowing the computational and the real world to influence the behaviour of each other. One of the main characteristics of a pervasive environment is its *proactivity*, as the system is expected to anticipate the user's needs rather than simply react to inputs.

Pervasive systems are expected to make adequate decisions on behalf of the users, relying on contextual information and on the computational devices available in the surroundings. Wearable (or implantable) devices form one type of interface between users and the pervasive environment. A wearable device can be a smartphone or a special vest with embedded sensors for health monitoring of a person. They must be aware of the user's context, and adapt their behaviour to the state of the user. Several aspects must be considered, such as types of I/O interaction and the restricted capabilities of these devices. A detailed discussion on wearable devices is presented in [SS03].

Several studies have been devoted to the design of infrastructures for pervasive spaces. Gaia [RHC$^+$02] seeks to extend the traditional operating system concept to the control of the devices available in a pervasive environment, as though these were the resources available on a single computer, e.g. disk, memory, audio, thus providing a view of a meta-operating system. A fundamental concept in Gaia is the *active space* which is an extension of a physical space which contains physical objects, including electronic/computational devices. An active space introduces a context-based coordination that caters for a transparent interaction between its components, detecting and adapting them in an autonomous way. Typically, active spaces are room-sized, and need to provide central servers for running the middleware services, e.g. event and context services. An active space follows a component-based infrastructure which extends the *Model-View-Controller (MVC)* pattern to reflect the wealth of input, output and processing devices. Gaia recognises the importance of federating Gaia spaces, but this is not part of the core view of its meta-operating system for smart spaces. Support for the interactions between standalone active spaces was later proposed in [AMCRC04] through the notion of *super spaces*. This focuses on how the middleware services of independent spaces are integrated in collections of peer-to-peer and hierarchical active spaces.

ISAM [A$^+$02] addresses resource management and application adaptation in large-scale environments. The ISAM architecture provides a pervasive environment which has a cellular organisation. Cells can be interpreted as institutional boundaries in a multi-institutional environment similar to virtual organisations in grid computing [FKT01], thus considering a coarser-grain notion of *space* compared to Gaia. An execution cell knows a set of other cells, which form its *neighbourhood*. However, ISAM relies on an *administrator* to inform the neighbours of each cell, which remain static for most of the time. ISAM also relies on a centralised server to host the services of an entire execution cell (discovery, scheduling, context service). These assumptions make ISAM unsuitable for ad-hoc environments.

One.world [Gri04, Gri02] focuses on an infrastructure that enables applications to follow users as they move through pervasive environments. One.world concentrates on three requirements of pervasive applications: *(1)* the system needs to adapt its behaviour explicitly to contextual changes instead of hiding it from applications; *(2)* the system needs to discover new resources dynamically instead of assuming a static environment; and *(3)* the system needs to facilitate sharing of information between applications and devices. One.world mainly relies on discovery, event and migration services. The devices visible on the local broadcast network are responsible for electing a centralised discovery server. An *asynchronous event service* provides the basic communication means, for both local and remote communication. The migration service is used to move or copy applications to different devices (the application's entire state is moved, including execution state and persistent data). One.world relies on *environments*, which are abstractions for structuring running applications, much like processes in an operating system in that both environments and processes represent an application being executed. Additionally, environments can be composed in a hierarchical way, providing a mechanism for extending applications where the outer environment has complete control over the inner environments. An outer environment can intercept and modify events sent by nested environments, providing an abstraction similar to nested processes in operating systems. Its implementation is based on nested *Java Classloaders*. There are a few similarities between environments and the composition of self-managed cells (as the outer environment having control over the inner, and the integration of event-based infrastructures). However, the scalability of One.world is limited, especially in relation to service discovery, making it suitable for small pervasive environments, with only a few dozen people and devices [Gri02].

iROS [JFW02, PJKF03] focuses on pervasive environments for meeting spaces. It emphasises the ability to integrate *legacy* applications, where they can be modified to be accessible in a standard way and customised to support different input/output interactions, for example, using speech input and multiple display output. In iROS the system is not expected to react to the user or context, and whilst the users must take responsibility for their actions, the system infrastructure is only responsible for providing a means of executing such actions instead of trying to adapt the environment automatically to changes in the context. iROS provides a message-passing infrastructure, which is centralised and broadcast-based and is limited to room-sized environments, which are termed *interactive workspaces*. iROS deliberately assumes a standalone pervasive space and does not provide support for the federation of multiple *interactive workspaces*.

Oxygen [SPP$^+$03, Oxy06] is a human-centred computing environment which is mainly concerned with the human-computer interaction aspects of pervasive environments, where the user should not be required to type or click but instead the system should be able to understand more natural forms of interaction, e.g. human speech, gesture or even lip movements. Oxygen presents the concept of *collaborative regions*, which are defined as self-organising collections of computers and/or devices that share some degree of trust. Such collaborative regions can be local-, building- or campus-wide, and they support multiple communication protocols for low-power local or wide area communication as in the SMC. Collaborative regions are formed to support the execution of specific tasks automatically, for example, recording and archiving speech and video fragments during a meeting (using a policy or script-based pre-configuration). These high-level user goals are satisfied by assembling generic standalone components that implement high-level functions (a voice recognition component, for example). These components are composed according to their interfaces. They are constantly monitored and if required, macro-level adaptation can be performed, substituting entire components. In essence, applications are represented as a graph of connected modules. Although Oxygen recognises the importance of establishing collaborative regions for executing specific tasks it is not clear whether collaborative regions interact with each other and how they can be federated.

These research studies address different aspects of pervasive computing, and it is rather difficult to directly compare their functionality and assumptions. However, they tend to share two limitations: they typically fail to provide a systematic way of composing and federating independent pervasive spaces into larger and more complex interactions, and they often focus on pervasive spaces of a relatively fixed size, e.g. a room, relying on a particular kind of infrastructure which does not permit the system to scale down to smaller spaces and scale up to form larger interactions. In contrast, the SMC is intended as an architectural pattern applicable at different scales, ranging from small body-area networks, to large virtual organisations. SMCs are expected to discover other SMCs dynamically, and support means of systematically specifying and establishing peer-to-peer collaborations, federations and compositions of larger structures.

2.1.2 Autonomic Computing

The other aspect of our work is autonomic computing, which advocates that the management of hardware and software systems cannot be reliant on human intervention, which is too expensive, error-prone and will not be able to cope with the scale of emerging pervasive systems. Instead, the system must be autonomous and capable of managing itself. The term *"autonomic computing"* was coined by IBM, and is an analogy to the autonomic nervous system of the human body which frees our conscious brain from the burden of controlling the body's vital functions and internal organs [Hor01, GC03].

The need for managing computing systems is not new, however the scale and complexity of pervasive and ubiquitous systems makes manual management or central coordination unworkable. Self-management thus enables the system to *configure* (add new components and functions dynamically), *optimise* (adjust system parameters), *heal* (detect, diagnose and repair failures) and *protect* (defend against attacks) itself in an autonomous[1] manner [KC03].

The industry work on autonomic computing, led primarily by IBM [KC03] but also addressed by Motorola [SAL06] and HP [HP03], focuses on network management of large clusters and web servers. The IBM autonomic manager has some similarity to the SMC approach in that it autonomously manages a set of resources, while exposing a management interface to other autonomic managers, as though it is a managed resource itself [BBC$^+$03]. However, there is

[1] In this thesis the terms *"autonomic"* and *"autonomous"* are used interchangeably.

no prescribed mode of operation for the specification or instantiation of interactions between autonomic managers. In addition to *monitoring* events and *executing* actions, the control-loop enforced by the autonomic manager is also responsible for *analysing* what is monitored and *planning* which actions should be taken. This control-loop relies on a *knowledge base*, which can grow dynamically as the autonomic manager learns about the system (Figure 2.1). The autonomic manager describes the functional aspects of behaviour present in the control-loop, but not a common architectural view or infrastructure that defines how these components should be realised.

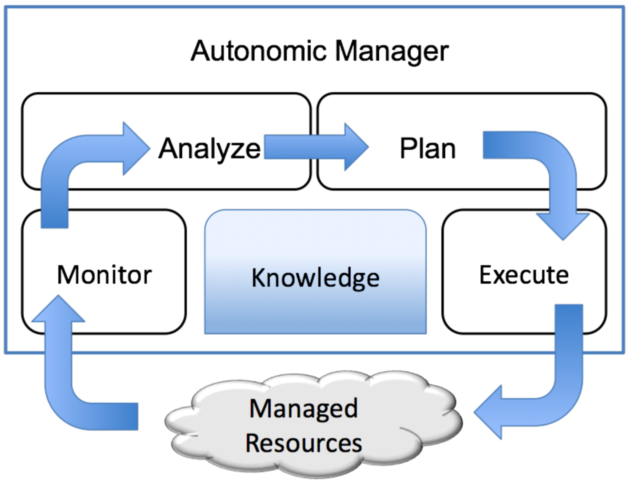


Figure 2.1: IBM autonomic manager's control-loop

Several research studies also advocate the use of planning techniques to address self-management in autonomous systems [SHMK08, KM07, GCH$^+$04, AHW04, OGT$^+$99]. The *Self-Managed Cell* does not rely on planning services or on the use of a knowledge base, although the SMC functionality can be extended to include these additional services as required. Instead, the adaptation strategy used in SMCs is based primarily on *policy-driven control-loops*. Planning does not scale as well as policies, since its generation is computationally expensive, and plans require increased specification effort for defining the domain model. Instead, writing policies is less cumbersome and they can be rapidly loaded to change (parts of) the adaptation strategy without interrupting the SMC's functioning. This is a fundamental characteristic in permitting the SMC to scale down to small resources as well as to scale up into more complex systems.

2.2 Policy Management

We advocate the use of policies for realising autonomous behaviour. Policy-based management relies on the use of rules to specify the management aspects of a system. Damianou and colleagues [D$^+$01] define a policy as *"a rule that defines a choice in the behaviour of a system"*. Policies separate the management strategy from the implementation of the management actions, which permits the modication of the run-time adaptation strategy without interrupting the operation of the system.

Policies can be applied in a number of areas, from adaptation of mobile devices into limited modes of operation, to management of resources in large-scale environments. According to Ganek and Corbi [GC03], policies will be used by autonomous systems not only to define management and authorisation functions, but also for quality of service, storage backup and general system configuration. Research on policies has been active for several years, especially policies for network and systems management. Examples include PCIM [MESW01], PDL [LBN99] and PMAC [ACG$^+$05]. Frequently policy-based management frameworks rely on infrastructured organisation models, e.g. using LDAP servers for implementing policy repositories [HCP07, HMP06]. Although they use policies that are similar to the ones we advocate for encoding adaptation, these approaches are targeted for the management of large-scale and networked systems, and do not scale down for managing small devices.

2.2.1 Types of Policies

Policies typically either require activities to be performed (*obligations*) or give authority to carry on a given activity (*authorisations*). For the purposes of this discussion, obligations and authorisations are defined as follows.

Obligation policies are *event-condition-action* rules of the form:

$$\begin{array}{l} \textbf{on}\ \langle event\rangle\ \textbf{do} \\ \quad \textbf{if}\ \langle conditions\rangle\ \textbf{then} \\ \qquad \langle target\rangle\ \langle action\rangle \end{array}$$

Obligations cater for the adaptive behaviour of the system and specify which actions must be performed in response to events, provided conditions are satisfied. The event is a term of the form *e(a1,...,an)*, where *e* is the name of the event and *a1,...,an* are the names of its attributes. The condition is a boolean expression that checks local properties of the system enforcing the policy and the attributes of the event. The policy designates a *target* upon which the action will be invoked. The action is a term of the form *a(a1,...,am)*, where *a* is the name of the action and *a1,...,am* are the names of its attributes. The target must support an operation that implements this action. The attributes of an event can be used for evaluating the condition, or as arguments to the action. Implicitly an obligation has a *subject*, which is the entity in charge of enforcing the policy. The target may be the same as the subject, i.e. actions can be performed locally.

Authorisation policies are access control rules of the form:

$$\begin{array}{l} \textbf{auth[+/-]}\ \langle subject\rangle\ \longrightarrow \textbf{if}\ \langle condition\rangle\ \textbf{then} \\ \qquad\qquad\qquad\qquad \langle target\rangle\ \langle action\rangle \end{array}$$

Authorisations specify which actions a *subject* is permitted (*positive authorisation*) or forbidden (*negative authorisation*) to invoke on a *target*, provided conditions are satisfied. The action and conditions have a similar format to those defined in obligations. The system in charge of enforcing an authorisation is typically the *target* of that policy, as it is usually assumed that targets normally wish to protect themselves from unauthorised actions. However, policies could be devolved to authorisation decision points if required.

2.2.2 Policy Conflicts

When policies originating from different systems are combined conflicts may occur if two or more policies apply to the same object. Policy conflict analysis processes are critical and have to be scalable and cope with the dynamism, heterogeneity and size of autonomic management infrastructures [DJS08]. Lupu and Sloman [LS99] present two main classes of policy conflicts: *modality conflicts* and *application-specific conflicts*. Modality conflicts arise when two or more policies with different *modalities* (positive and negative) refer to the same subject, action and target. For example, a subject may be at the same time authorised and forbidden (by both a positive and a negative authorisation) to perform an action on a target, or a subject may be required (by an obligation) but forbidden (by a negative authorisation) to perform an action on a target. Typically, modality conflicts can be identified through syntactic analysis of the policies, and precedence schemes are often used for automatically solving such conflicts. Several research studies address policy conflicts from the perspective of detecting overlapping subjects, targets and actions [UBJ+03, RDD07, KFJ03]. Conflict detection based on situations where the condition part of multiple policies may be simultaneously true, i.e. two policies become applicable and may specify two incompatible actions, has been studied in [AGLL05].

Application-specific conflicts occur if what is contained in a policy is inconsistent with specific concepts and semantics related to the application domain. These involve *separation of duty conflicts* [CW87] (where the same manager performs two or more tasks that are supposed to be performed by different managers) and *conflict of interests* (where the same manager manages two or more objects that are supposed to be managed by distinct managers), for example. Application-specific conflicts cannot be automatically identified through syntactic analysis, and they typically require the use of *meta-policies* to specify constraints that provide additional information for conflict resolution. Frameworks for policy analysis and detection of application-specific conflicts are presented in [BLR03, CFP+06, CLM+09].

2.2.3 Policy Refinement

User goals are typically expressed as high-level policies that are not directly implementable. Policy refinement provides support for translating abstract goals into concrete, implementable policies that can be enforced by the system. The main objectives of policy refinement are: *(a)* determine the resources which are required for fulfilling the policy requirements, *(b)* translate the high-level policies into concrete policies that the system can enforce, *(c)* verify that the lower level policies still meet the high-level goals [Ban05]. This ensures that the implementable policies achieve the same functionality intended by the high-level ones.

2.3 Self-Managed Cell Framework

The *Self-Managed Cell* (SMC) [LDS+08, SBD+05, DLS+05, DHL+05, S+06] framework provides an infrastructure for building autonomous ubiquitous systems. It relies on the use of policies as the principal mechanism for achieving autonomous behaviour and has evolved from previous work on policy-based management developed at Imperial College [Slo94b, D+01, LSDD00, DLSD01, SL02]. An SMC consists of a set of hardware and software components that are able to work autonomously, based on a policy-driven feedback control-loop that determines which management or reconfiguration actions must be performed in response to changes in the SMC context.

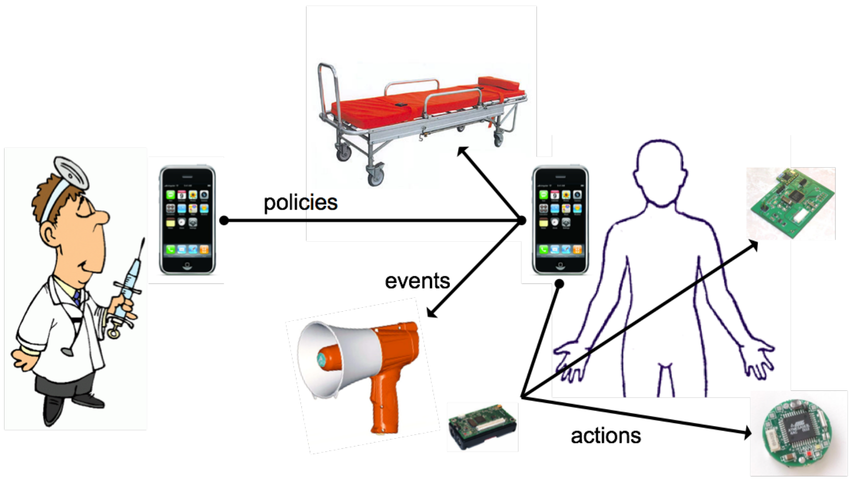


Figure 2.2: Body-area SMC

A typical SMC used for health monitoring (Figure 2.2) comprises a smartphone or a *Gumstix*[2] device hosting SMC management services that control several sensors such as heart rate, temperature, acceleration, blood pressure and oxygen saturation hosted on *body-sensor nodes*[3] (BSN) which may be SMCs themselves. Actuators such as a pacemaker or an insulin pump SMC are activated according to conditions monitored by the sensors, or alarms are set depending on the measurements made by these sensors. Smartphones or other *Gumstix* devices are also used to host application services, e.g. a diagnostic service hosted in a remote device. An SMC running on a doctor's or nurse's smartphone can be used to interact with the patient's body-area network, either prescribing how monitoring must be conducted, or collecting its results. Communication with BSN nodes typically occurs through IEEE 802.15.4 radio links while communication between *Gumstix* devices or with smartphones occurs through Bluetooth or Wi-Fi.

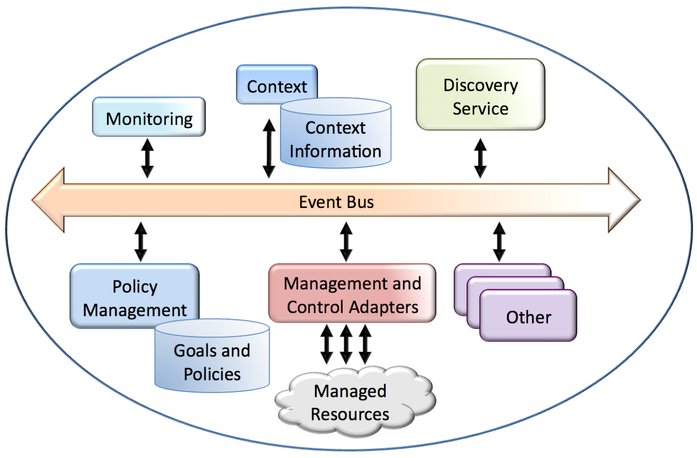


Figure 2.3: Self-managed cell architecture

We regard the SMC as an architectural pattern, where the SMC can be instantiated to individual devices, as well as scale up to cater for the management of larger ubiquitous systems. The SMC framework relies on a dynamic set of *management services* (Figure 2.3): an *event bus* that is used to carry events between the various components and services within the SMC, a *policy service* that enforces adaptive and access control rules, and a *discovery service* that can discover new components which are able to join the SMC. Typically dif-

[2]http://www.gumstix.com
[3]http://vip.doc.ic.ac.uk/bsn/

ferent implementations of these services would be used on small-scale devices and on large-scale environments. The architecture is *extensible* and additional services, e.g. for retrieving a specific type of contextual information, can be added to extend the SMC architecture as required [LDS+08].

The Self-Managed Cell resembles a *sentient object* [GBCC02, VCC+02, BC04] in that both are intended to model a set of hardware and software components that interact with each other, and provide an infrastructure to support large-scale distributed systems composed of mobile autonomous components. Both rely on adapters to abstract low-level communication protocols, and use an event-based infrastructure to support event dissemination among collaborating components. A sentient object accepts input via its sensors, and reacts by acting upon the environment using actuators. Actuation decisions are defined in a CLIPS (C Language Integrated Production System) rule-based inference engine. However, the sentient object model differs from SMCs in three important aspects:

1. SMCs can discover each other and load new policies (as a *mission*) to define how remote SMCs must interact. Sentient objects do not exchange policies, and each sentient object has a static set of production rules defined in CLIPS which are used to infer other rules or generate events.

2. Their aims are different: sentient objects are intended for collaborative inference of actuation decisions. In contrast, SMCs are used to enforce the management actions and to distribute management policies dynamically among remote SMCs.

3. Collaborating sentient objects follow a *WAN-of-CANs* [VCC+02] structure (wide-area network of smaller, local controller-area networks). However, this model does not support the dynamic assembly of sentient objects using more general and reusable patterns of interaction, such as the ones presented in this thesis.

The SMC management services are discussed in detail in the next sections.

2.3.1 Event Bus

A publish/subscribe *event* bus is used to provide the underlying communication infrastructure within an SMC, although it is not required that all communications happen through events. This has the advantage of de-coupling the services and resources that are part of an SMC, as an event publisher does not need to have prior knowledge of the recipients when sending a message. This also permits adding new services to the SMC without disrupting existing ones. The event bus can be used for both management and application communication, such as alarms indicating that thresholds have been exceeded.

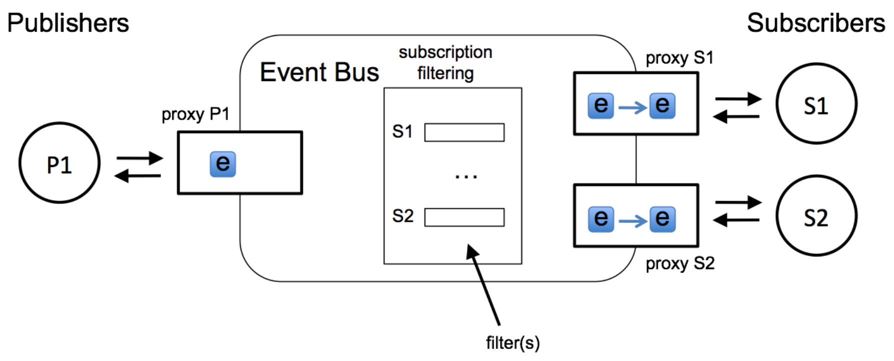


Figure 2.4: SMC event bus

The event bus implements a content-based delivery service, where event matching can be performed over any field of the event message. Messages are routed by the event bus using *filters*, which match the subscriptions with the content of the events published (Figure 2.4). While the *subscribers* of an event must specify their needs by explicitly registering themselves with the event bus, the same does not apply to event *publishers*. This reduces the coupling of the system and requires less information to be stored in the event bus service.

In [SB08, SVBM08], a publish/subscribe system tailored to the healthcare domain is presented, where personal data is highly sensitive and fine-grained control for data transmission is needed, which may involve for example the patient consent and context information, e.g. emergency situation. This uses transformation policies to regulate information flow on a *need-to-know* basis, and is built into a PostgreSQL database management system. Our event bus is less sophisticated as the SMC is intended to run both on resource-constrained devices as well as on larger systems, but new services can be added to the SMC to extend its functionality, e.g. a context service or event correlators.

2.3.2 Policy Service

Our main SMC implementation relies on a policy service which is based on the *Ponder2* policy framework[4] (but we also provide a lightweight implementation that can run on BSNs and other constrained devices [K$^+$07]). It implements a policy execution framework that supports the enforcement of both obligation (*event-condition-action*) and authorisation (*access control*) policies. Policies can be dynamically loaded, enabled, disabled and unloaded to change the behavior of the SMC without interrupting its functioning.

Ponder2 comprises a general-purpose object management system. The policy service maintains *managed objects* for each of the components on which management actions can be performed. This includes sensors and local resources that the SMC owns, services within those devices, and adapters for remote SMCs. Managed objects representing remote SMCs adapt invocations received to platform-specific actions, thus providing a uniform interface for accessing SMC services. Managed objects are kept in a *domain structure* [TS88] that implements a hierarchical namespace within the SMC, similar to a file system. A *domain* provides a way of grouping objects for the purposes of policy specification, as the specification of policies in terms of a large number of individual subjects and targets is impractical [Slo94a]. Grouping of objects can be based on their type, management functionality, or simply convenience.

Policies are written in terms of *managed objects*. Domains and policies are managed objects in their own right on which actions can be performed; for example, adding or removing an object from a domain, or enabling or disabling other policies. Thus, events can trigger obligation policies that enable or disable other policies, allowing the SMC to modify its own adaptation strategy during

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

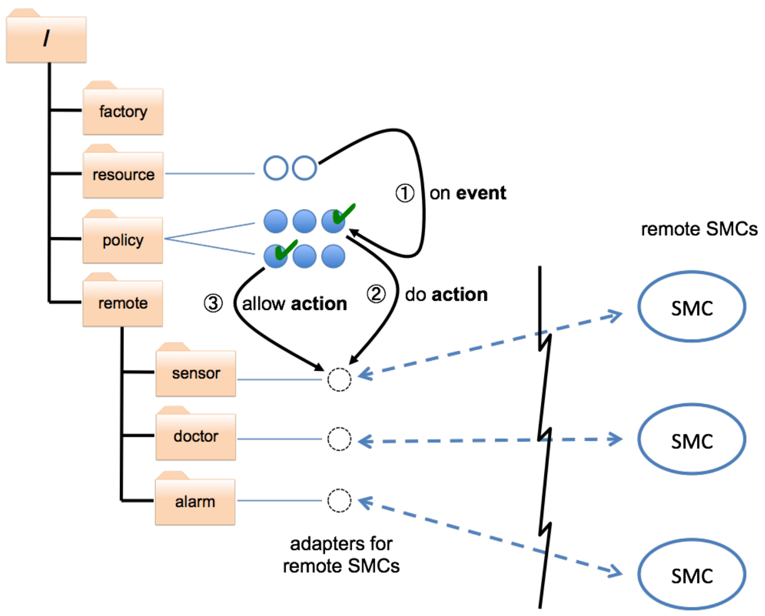


Figure 2.5: SMC domain and managed objects

run-time. Policies specified in terms of a specific domain will apply to all objects inside that domain. The set of policies an SMC enforces will prescribe how its managed objects should behave during SMC functioning. Thus, events generated by managed objects within the SMC trigger obligation policies, which specify what management actions must be invoked in other objects, provided these actions are allowed by corresponding authorisation policies (Figure 2.5). Management actions are invoked in remote SMCs via the adapters stored in the local domain.

The policy service has been designed with particular focus on flexibility, in that all the code needed can be loaded on demand. This is done via *factory objects*, which are managed objects themselves and permit the creation of new objects in the domain. Factories can be loaded dynamically and used for creating new policies, for creating domain objects to form a hierarchy, for creating adapters to communicate with external SMCs, and for creating event templates to communicate with the event bus. In particular, the latter enables the policy service to subscribe to specific event notifications, which are used by the policies to trigger adaptive actions on other managed objects.

2.3.3 Discovery Service

Resource discovery provides a means for automatically locating devices and services in a network [FDC01]. The SMC *discovery service* is used to detect new devices capable of joining an SMC, e.g. sensors or other SMCs in the vicinity. The discovery service interrogates newly discovered devices to establish a profile describing the capabilities they offer, and then stores a managed object as a reference to the discovered resource in the domain structure of the discoverer SMC. This managed object works as an adapter that abstracts the communication protocol, e.g. sockets, RMI, HTTP, between the discoverer and discovered SMCs. All the policies applying to the specific domain where the managed object was stored will then apply automatically to the discovered resource itself. An event describing the addition of a new device is generated, so other components within the SMC can use the new resource as appropriate.

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. The discovery service is also capable of detecting when one of the SMC's resources has left, distinguishing transient failures, which are common in wireless communications, from permanent departures, e.g. device out of range, switched off, or failure. 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. An event describing the departure of one of its current components is then generated within the SMC, allowing other components to adapt their behaviour accordingly.

2.3.4 Feedback Control-Loop

The SMC's event bus, policy and discovery services implement a control-loop which enables the execution of adaptive management actions in response to changes in the SMC's context. The control-loop operates as follows: managed resources within the SMC signal the occurrence of a particular event. Obligation policies, in the form of *event-condition-action* rules, are used to specify what actions are to be executed when a given event occurs (provided the policy's condition is satisfied). The action prescribed by the policy is then executed to adapt the operation of the SMC's resources with respect to the event initially detected, e.g. to adjust the monitoring frequency in a patient SMC in response to an event indicating a high heart rate measurement (action enforcement is itself dependent on the existence of authorisation policies). This in turn may cause the managed resources to generate new events, which will trigger new policies (Figure 2.6). Components in an SMC can change dynamically, e.g. a sensor may be added, or a device may fail, and these circumstances also require adaptive or reconfiguration actions to be performed.

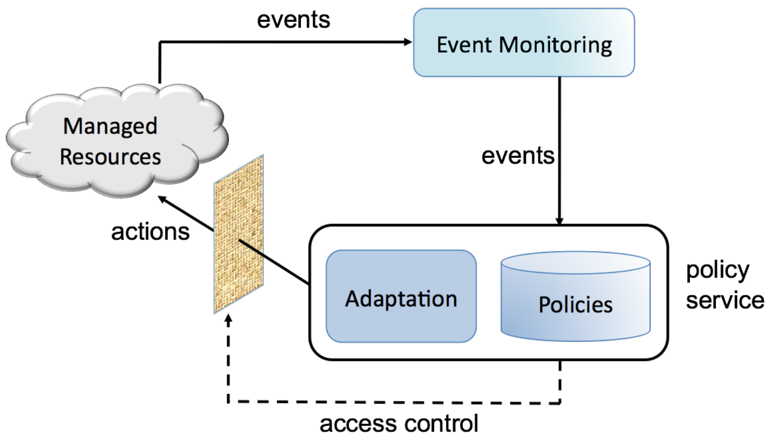


Figure 2.6: Policy-driven feedback control-loop

Although individual SMCs are autonomous and enforce their own feedback control-loop, applications typically involve a large number of interacting SMCs which must be composed and federated in complex ways. Thus support is required for facilitating the specification, instantiation and reuse of SMC interactions.

To realise the systematic construction of policy-based SMC interactions we adapted and incorporated concepts from different research areas. In the following we discuss techniques advocated for structuring the development of software, component-based systems and multi-agent systems, and identify how these can benefit the engineering of policy-based interactions between SMCs.

2.4 Component-Based Systems and Design Patterns

Collaborations between SMCs will likely involve the combination of a number of smaller interactions for realising more complex applications. As will be discussed in Chapter 3, interactions between SMCs are based on very specific abstractions for defining how the interaction is organised, how roles are assigned and policies are exchanged, and how events for triggering these policies are shared among the SMCs. The specification of SMC interactions can benefit from *component-based systems* and *software architectures*, which have provided various means of organising software development. Moreover, the notion of *design patterns* advocates the identification of reusable design solutions for recurring problems in software development. This is of particular importance to SMC interactions, as we seek means of systematically composing and reusing simpler abstractions for building more complex policy-based interactions. This section presents a brief literature review on these research areas.

2.4.1 Components and Software Architectures

Software architecture can be seen as a set of principal design decisions made during the conceptualisation and development of a system [TMD09]. The software engineering community has long investigated software architecture-based approaches, which typically separate computation (*components*) from interactions (*connectors*). Components and connectors are defined as follows [TMD09]:

- *Component*: architectural element that encapsulates functionality and data. It provides access to these via an interface, and has explicit dependencies on its required execution context.
- *Connector*: architectural element responsible for regulating the interactions among a set of components.

A component thus represents the computation and state of a system. It provides functionality to other components but also requires functionality provided by other components in its context. A connector facilitates the interaction between a set of components. While components usually support functionality that is specific to an application, software connectors are often application-independent and thus can be used across applications repeatedly (yet, they are not always considered a first-class entity in many examples of software architectures) [TMD09]. The benefits brought by the distinction between components and connectors have been widely recognised in the software community as a means of structuring software development [GS93, SDK$^+$95, SDZ96, MMP00, MT00]. The separation of computation from interactions itself dates back to the research on *programming-in-the-large* versus *programming-in-the-small* [DK75]. Although components and connectors do not cater for the adaptive behaviour of SMCs as expressed in roles and policies, similar principles can be applied for structuring and reusing SMC interactions to form larger collaborations.

Software architectures often use architectural styles, and patterns, to design and specify the overall structure of a system. The distinction between styles and patterns is traditionally fuzzy and it is not always possible to define a clear boundary between them [TMD09]:

- *Architectural style*: high-level architectural decisions that are applicable in a given context, constraining the architecture of a particular system while highlighting the benefits achieved by those decisions.
- *Architectural pattern:* more specific design decisions (usually from the refinement of a style) in order to be applied to a particular system. These can be thought of as architectural styles that are instantiated with the components and connectors that are pertinent to a given application.

Common examples of architectural styles are *client-server*, *blackboard* and *pipe and filter*. Figure 2.7 shows an example of the pipe and filter architectural style, where a filter component receives data from an input pipe, transforms this data and sends data to the next component via an output pipe. The main characteristic of this style is that filters are independent from each other, and do not have knowledge of other filters that come before or after them. Moreover, the correctness of the output of the network of pipes and filters should not be influenced by the order of the filters in the pipeline [GS93].

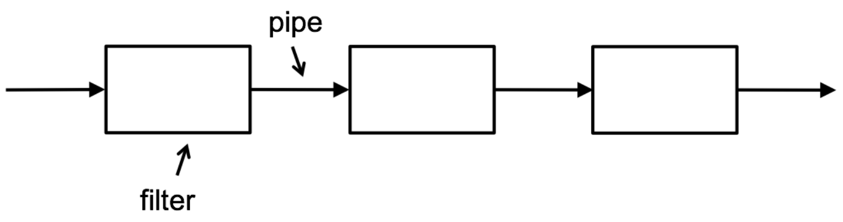
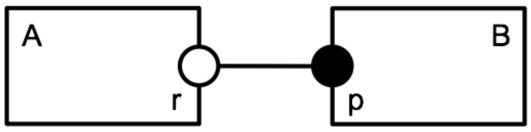


Figure 2.7: Pipe and filter architectural style

Systematic specification of SMC interactions can benefit from *architectural description languages (ADLs)*, *module interconnection languages (MILs)* and coordination schemas in general, such as Wright [AG94], UniCon [SDK$^+$95], Conic [Kra90], Darwin [MDEK95], Rapide [LKA$^+$95] and Mobile UNITY [RP03]. However, although traditional architectural description languages and coordination schemas can *bind* software components through connections, the semantics of these connections (such as *"sends data to"*, *"controls"* or *"is part of"*) is not always clear, failing to represent higher-level relationships between these components [Cle96]. An exception to this is the notion of *user-defined connectors* [ASCN03] and *higher-order connectors* [LWF01], which support the incremental building of more complex connectors from simpler ones.

Darwin configurations, for example, use *bindings* to link the *provided* functionality of one component's interface to the *required* functionality of another component's interface, but these bindings usually do not have any higher-level semantics associated with them, as connectors are not treated by Darwin as a first-class concept. Figure 2.8 shows an example Darwin configuration [MDEK95] between a client and server components (white circles represent required interfaces, black circles represent provided interfaces). In this example, a system is defined by instantiating a client component and a server component, and binding the required interface *"r"* of the client to the provided interface *"p"* of the server.



```
component Server {          component System {
    provide p;              inst
}                               A: Client;
                                B: Server;
component Client {          bind
    require r;                  A.r -- B.p
}                           }
```

Figure 2.8: Darwin configuration

Self-Managed Cells are similar to components and SMC interactions can be designed in a manner akin to software connectors that exhibit well-defined properties for binding SMCs. In contrast to architectural description languages however we are not interested in general-purpose component interactions, but instead aim for a model that addresses the structuring of policy-based collaborations using the SMC infrastructure.

2.4.2 Design Patterns

Design patterns are reusable solutions for recurring problems in software design [GHJV95]. Object-oriented design patterns, for example, address the limitations of pure object-oriented design techniques, which are not well suited for describing complex interactions between groups of objects or classes [MKMG97]. The design patterns presented in [GHJV95] are divided into three categories: *creational patterns*, which provide mechanisms for object creation, e.g. *Factory, Prototype, Singleton*; *structural patterns*, which support different ways of realising relationships between entities, e.g. *Bridge, Decorator, Composite*; and *behavioural patterns*, which provide patterns that increase the flexibility in carrying out the communication between objects, e.g. *Command, Observer, Mediator.* These design patterns provide general templates on how to solve a problem in different situations, without specifying the actual classes or objects that will be involved in the final application. The use of design patterns promotes a better understanding of the software itself, and the reuse of well-understood abstractions.

Multiple design patterns can be applied simultaneously to the design of the same application. Each design pattern can thus define a different aspect of the interaction between a group of classes. The composition of design patterns [Don02, Don03, Don04], where the same class plays different roles in the context of different design patterns, is analogous to an SMC playing different roles in distinct interactions.

Software architectures and design patterns are complementary to each other. Garlan and colleagues [GCK02] investigated the possibility of using object-oriented notations for the specification of architectural descriptions, and mapping ADLs into UML notation. However, it is claimed that there is no single best way of encoding one into another because of semantic mismatches between the two. Moreover, Monroe and colleagues [MKMG97] also observe that architectural patterns and object-oriented design patterns can be seen as complementary techniques for system design, where the former is concerned with the coarse-grained composition of components and their interactions whereas the latter can be used to refine the internal implementation of a more sophisticated component or connector.

Finally, specific design patterns for pervasive systems are starting to emerge [LB03, CHL$^+$04], but these so far have focused on high-level HCI aspects of these interactions, addressing patterns of user/computer interaction within pervasive environments. An example of such a pattern is the *context sensitive I/O pattern*, where the mobile phone detects that the owner is driving or in a meeting and accordingly either rings or vibrates. Instead, our work on SMCs concentrates on the management aspects of the system.

2.5 Multi-Agent Systems

The work on multi-agent systems has investigated means of organising autonomous collaborating entities into more complex aggregates, whose principles and techniques can also be applicable for constructing cross-SMC interactions. In its most general sense an *agent* denotes a hardware or software system that presents the following properties: *autonomy* (operate without human intervention), *social ability* (agents interact with each other), *reactivity* (perception of the environment and subsequent response to it) and *pro-activeness* (not only react but also perform goal-directed behaviour) [WJ95]. This is considered a *weak notion* of agency, usually adopted in agent-based software engineering. Under this perspective, UNIX processes or software components that exhibit the aforementioned properties can be seen as agents.

A *stronger notion* of agent is the one where an agent exhibits, in addition to the above properties, an anthropomorphic behaviour relying on concepts and attributes that are more usually applied to humans (for example, *beliefs, desires, intentions* [RG95]). In the artificial intelligence community, this corresponds to an *intelligent agent* which is usually also associated with the ability to reason and learn, and how these techniques can be used by the agent to interpret and use the knowledge it has access to. Intelligence varies from marginal intelligence (e.g. expressed as preferences) to advanced intelligence (e.g. derive new knowledge via learning techniques) [CH97]. The Self-Managed Cell concept is based on a feedback control-loop that relies on policies to govern its adaptive behaviour and interactions with other SMCs, which does not involve learning or reasoning (although these techniques could be used to extend the SMC's functionality) and is therefore closer to the notion of weak agency.

A *multi-agent system* (MAS) [Woo02] is composed of multiple interacting agents that need to cooperate, coordinate and negotiate, in order to solve problems that could not be solved by individual agents alone. Agent-oriented software engineering has recognised the need for *organisational structures* for designing multi-agent societies. In general, multi-agent systems focus on a *role model* for agents, however more than a simple collection of roles, complex systems require further organisational abstractions for assisting their design and analysis [ZJW03]. Expressing organisational structures explicitly enables the construction of a system in a robust and repeatable fashion. Holonic models [HKR08, RHK03, UG02] often support hierarchical structures, but not more sophisticated organisations. A few multi-agent systems have attempted to identify a comprehensive catalogue of generic and reusable patterns for interactions between agents, which express pre-defined and common organisational structures that could be reused across different systems [ZJW01, ZJW03, HCY99, TOH99, KKPS98]. In addition to the organisational structure of SMC interactions, there is also a need to define how SMCs interact with each other in such aggregated groups to distribute management responsibility, application tasks or to implement communication patterns.

Deugo and colleagues [DOKvO99] present patterns for mobile and intelligent agents, and patterns to manage the communication between agents, e.g. creating a proxy in its home location when the agent moves away to hide its change in position, or a session pattern for managing complex conversations between multiple agents over a period of time. These patterns highlight the importance of engineering complex multi-agent systems and a methodology for pattern choice, which specifies the *context* where a pattern should be applied and the *forces* that influence or constrain the choice of a specific pattern. Aridor and Lange [AL98] propose specific patterns for mobile agent applications concentrating on issues related to the agent's ability to move from one machine to another. These include travelling patterns used for managing the movement of mobile agents, e.g. patterns for specification of an agent's itinerary involving multiple destination hosts or patterns for moving groups of agents that must travel together, as well as patterns for coordinating the meeting with other agents when arriving at a specific destination. Examples of mobility patterns proposed in [TOH99] are illustrated in Figure 2.9. These are called *itinerary*, where an agent iterates around several destination hosts performing tasks in each one, *star-shaped*, where agents move back and forth between the necessary hosts, and *branching*, where an agent generates copies according to the number of hosts it needs to visit. Kolp and colleagues [KGM02] propose a macro-level catalogue for interactions between agents for business process by using real world business organisations as an analogy, e.g. structure-in-5, pyramid, bidding, joint-venture (agreement between partners), etc.

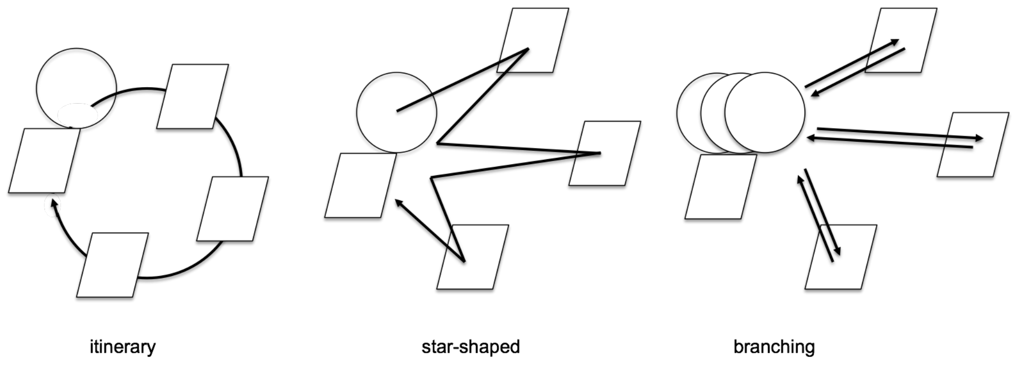


Figure 2.9: Agent mobility patterns

However, most of the work in multi-agent systems focus on a subset of problems known as *distributed problem solving* [Smi88]. This corresponds to the co-operative solution of problems by a decentralised collection of agents that possess different knowledge sources, and need to delegate tasks to other agents that might be more suitable for solving the problem or parts of it. None of these efforts focuses on patterns for engineering autonomous policy-based systems. It is thus needed to adapt these general principles to the requirements of the SMC architecture, as we are interested in patterns for systematically composing and federating SMCs in complex collaborations but concentrate on the policy-based aspects of these interactions.

2.6 Discussion

This chapter started by presenting the more general related work on pervasive and ubiquitous systems, and how devolved management is a promising approach to tackle the complexity of large-scale ubiquitous systems, where autonomous components endowed with self-adapting capacity are able to effect local decision-making behaviour. We then presented the background work on policy-based management and on the Self-Managed Cell framework, which advocates the use of policies as the principal mechanism for achieving autonomous behaviour. The SMC relies on a policy-based feedback control-loop to perform adaptive actions in response to changes in its context, or changes in the context of its managed resources. Using its discovery service an SMC is able to discover other resources that are capable of joining the SMC. Discovered resources are stored in a specific domain within the SMC, and policies written for that domain will then automatically apply to resources assigned to it. The services and resources within the SMC communicate via an asynchronous event bus, which facilitates event exchanges among these components.

The SMC serves as an infrastructure for building ubiquitous computing applications. Although the SMC concept provides a suitable abstraction for representing autonomous components, applications will typically consist of ad-hoc collaborations of devices and resources and therefore require a large number of elementary SMCs to negotiate the necessary interactions with other components in their surroundings and be assembled into larger and more complex structures based on the same principles of self-management. We incorporate ideas from multi-agent systems, software engineering and software architecture principles, and apply these to describe policy-based collaborations of SMCs, in particular, techniques for *structuring* the software organisation, its *reuse* and the identification of recurring *patterns* or catalogues for composing software components and agents. Our overall goal is to use the SMC as an architectural pattern for building policy-based autonomous systems, where individual SMCs can be assembled systematically into larger and more complex structures. Unlike general purpose component models we need to adapt, design and implement the general principles to the specifics of the SMC architecture and its policy-based operational model.

Basic SMCs, such as the one representing a personal device or body-sensor, can be part of more complex SMCs, such as a body-area network or a home monitoring system, which form a collection of smaller, yet autonomous, SMCs. In the next chapter we describe the concepts that provide the underpinnings for facilitating SMC interactions. These will serve as the basis for building an infrastructure for the collaboration of policy-based autonomous systems.

Chapter 3

Basic SMC Interactions

This chapter discusses the elementary issues related to interactions between self-managed cells. We focus on the underlying principles that facilitate the establishment of interactions and describe how SMCs can provide functionality to other SMCs and how they can exchange policies that prescribe SMC behaviour within the context of an interaction.

3.1 Motivation and Requirements

Although SMCs are autonomous, applications typically require a large number of SMCs to collaborate, and SMCs must be able to interact with each other in complex ways, to federate or compose into larger structures. This chapter discusses basic interactions for collaborating SMCs in either peer-to-peer or compositional settings. We can draw several requirements from the example applications for healthcare monitoring and coalitions of autonomous UAVs presented in Section 1.1.

Firstly, SMCs must detect the presence of remote SMCs and decide autonomously whether to establish an interaction. Interfaces specify the operations one SMC provides to another, the events an SMC is able to receive from other SMCs and the selected internal events that an SMC can use to notify others. More complex interactions may require exchanges of policies between the SMCs, e.g. when an SMC can request another to behave in a specific way. Secondly, the interface exposed to a remote SMC may include only a subset of the available functionality depending on the kind of SMC and the role, e.g. doctor or nurse, it can play in the interaction. Finally, an SMC may wish to provide access to its internal resources to other SMCs and to mediate the access to those resources.

Whilst *peer-to-peer* interactions occur frequently as SMCs interact with neighbouring autonomous components, *composition* interactions enable grouping SMCs into larger autonomous structures and scaling SMC management to larger environments. Composition encapsulates an SMC with its own resources, as a managed resource within the containing SMC. This implies that the SMC can be programmed by the containing SMC in terms of policies that it must enforce. Moreover, the device exposes to its containing SMC a management interface for re-configuration. For example a diagnostic device may be part of a body-area network that will load new decision algorithms and new policies into it. Similarly, larger sensors may be autonomous components and thus SMCs in their own right.

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

Thus a composition differs from a peer-to-peer interaction in the degree of access permitted, i.e. which methods and events are exposed and which policies are accepted from the containing SMC, and in the fact that interactions between the contained SMC with external resources must be mediated by the containing SMC. However, compositions and peer-to-peer interactions have similar requirements in terms of the mechanisms required for supporting these interactions:

- Firstly, an SMC must expose functionality to interacting SMCs and facilitate access to its services or own internal resources, e.g. sensors in a body-area SMC. Thus an SMC must be able to offer an ***interface*** that specifies the SMC's functionality and which can be exchanged among remote SMCs in a collaboration.
- Secondly, an SMC must be aware of the state or context of their collaborators, as this may influence its own behaviour. Thus an SMC must be able both to receive ***events*** from other SMCs and also to notify remote SMCs of selected internal events occurring within itself.
- Finally, more complex interactions require an SMC to load new tasks dynamically into a remote SMC. This can be achieved through exchanges of ***policies*** between the SMCs, where an SMC can request another to behave in a specific way within the context of the interaction.

The exchanges of interfaces, events and policies provide the basic mechanisms for supporting SMC interactions. These will be discussed in detail in the following sections. In Chapter 4 we will describe how more complex interactions can be engineered in terms of *patterns of interaction*, which are based on the use of specific algorithms or protocols for realising these exchange mechanisms.

3.2 Interfaces

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 specification defines the operations supported by the SMC, as well as the events that can be sent and received to/from remote SMCs (Figure 3.1). Formally, an interface description i 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 can be published externally by the SMC (i.e. to which external SMCs can subscribe); and
- N is a set of *notifications*, which are external events of which the SMC can be notified (i.e. that external entities publish within the SMC).

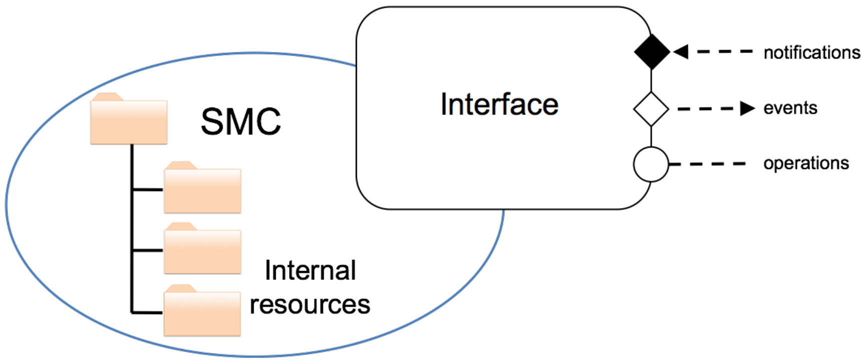


Figure 3.1: SMC interface

3.2.1 Core Interface

The SMC's *core interface* is application-independent and defines the functionality required for supporting the basic exchanges of events and policies, and for retrieving the interfaces that define the application-specific functionality which an SMC supports. The core interface is therefore required for enabling the establishment of SMC interactions in terms of the basic exchange mechanisms.

The *operations* supplied by the SMC's *core interface* are:

- *"notify"*: sends an event notification to a remote SMC;
- *"load"*: loads a group of policies into a remote SMC;
- *"unload"*: unloads a group of policies from a remote SMC;
- *"getInterface"*: retrieves a specific customised interface for a remote SMC.

Additionally, the *events* and *notifications* specified by the *core interface* define application-independent phenomena in the establishment of SMC interactions, namely the discovery/departure of SMCs, and the loading/unloading of policies:

- *"foundSMC"*: a new SMC has been discovered;
- *"leftSMC"*: an SMC has departed from the interaction;
- *"loaded"*: a group of policies has been loaded into the SMC;
- *"unloaded"*: a group of policies has been unloaded from the SMC.

3.2.2 Customised Interface

The management of an SMC's application-specific behaviour is defined through its *customised interface*. For example, in applications for healthcare monitoring, a patient SMC may provide an interface which allows the doctor to read sensor measurements or set new alarm thresholds. An SMC can support multiple customised interfaces, which allow different partner 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 application 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. Exposing the same functionality to all partners could thus have security implications in military applications or unwanted privacy implications for example in medical applications, as any patient SMC would be able to find out what other patients are up to. Typically, the customised interfaces an SMC exposes depend on the kind of SMC it is interacting with, e.g. a doctor and ordinary hospital staff will have different *"views"* of the functionality provided by a patient SMC, and may also depend on the specific instance SMC, e.g. the GP treating that patient will have access to more functions compared to any other GP. Thus, an external SMC should see only those functions that an SMC wants to expose in a customised interface generated specifically for that interaction.

Complex SMCs typically comprise internal resources and smaller SMCs. In this case, the functionality provided by the patient SMC's customised interface will rely on the functionality of its resources. This can be achieved by mapping the functions exposed in the interface of the containing SMC to operations supported by its internal resources, e.g., the patient interface may support a *"readTemperature"* operation, which is mapped to a specific operation in a sensor object that is part of the SMC's internal structure. Additionally, this permits controlling which methods are exposed by a specific customised interface by dynamically adding or removing the mappings.

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

Figure 3.2: Customised interface of a patient (Ponder2 XML notation)

Figure 3.2 illustrates the customised interface of a patient SMC: it defines the events, notifications and operations supported by that interface. This interface in particular is able to generate the event *"monitoringReady"*, and be notified of the events *"startMonitoring"* and *"stopMonitoring"*. The interface supports the operations *"readECG"* and *"scheduleTask"*, which are mapped to the functionality provided by local resources within the SMC. Events and notifications have an external name, but these are mapped to specific local events defined within the context of that SMC.

3.3 Interaction Establishment

The establishment of a new interaction between SMCs 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 classify and determine how the interaction will occur; and *(3)* a policy-based interaction between the SMCs is set up. The first two steps of the process are detailed below, and the third step is discussed in Section 3.4.

3.3.1 SMC Discovery

Interaction establishment is initiated as a result of the discovery service of an SMC (Section 2.3) detecting the presence of another SMC. An SMC's discovery service broadcasts its *identity message* (id;type[;extra]) at frequency R, where:

- *id*: the address of the SMC, e.g. rmi://gumstix4.doc.ic.ac.uk/smc;
- *type*: the type of the SMC, e.g. doctor, patient;
- *extra*: additional information about the SMC, such as *capabilities* supported by the SMC, e.g. heart rate monitoring, and *credentials* that the SMC possesses, e.g. public-key digital certificate.

When a remote SMC is detected, the discoverer SMC generates the event *found-SMC* within its local event bus [LDS+08]. The event contains the information broadcast by 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 *core interface.*

The patient provides one interface to a doctor, via the core interface, which is possibly different from the interface provided to a nurse (Figure 3.3). When an SMC provides an interface to other interacting SMCs, 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. This may in turn cause the discovered SMC to obtain a customised interface for the discoverer. An SMC may need to authenticate its partner using the *extra* information supplied in the identity message before handling a specific customised interface to the remote SMC, e.g. a malicious SMC could pretend to be a doctor in order to acquire an interface for the patient SMC.

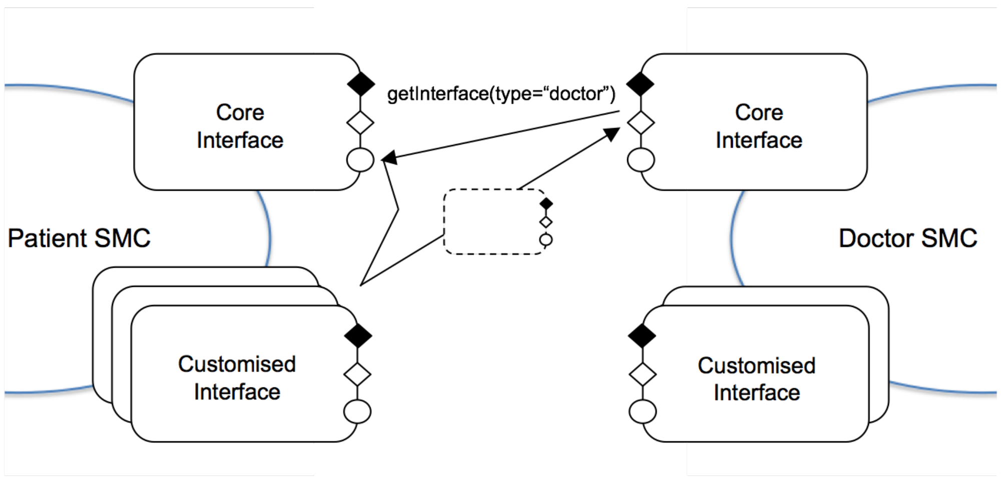


Figure 3.3: SMC interface exchange

3.3.2 Role Assignment

The use of *roles* for structuring responsibility in the context of distributed systems management has been thoroughly discussed in [Lup98, LMR98, Lin01, San96, SCFY96, SBM99, HBM98, HYBM00]. We use *roles* as placeholders for remote SMCs that are discovered at run-time, and these placeholders are kept within the local domain structure of each SMC. Each role is associated with a specific behaviour that can be performed by a single SMC within the context of an interaction.

When an SMC is discovered and a customised interface for the interaction is obtained, the role this SMC will be playing within the context of the interaction is determined based on the *type* and/or *capabilities* of the discovered SMC, i.e. doctor SMC will be assigned to a role for doctors, whereas a patient SMC will be assigned to a role for patients (Figure 3.4). A 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 $Interface_c = \langle O_c, E_c, N_c \rangle$ be the customised interface provided by a discovered SMC, let r be a role defined within the discoverer SMC, and let $Interface_r = \langle O_r, E_r, N_r \rangle$ be the expected interface for that role, then the assignment of the SMC which provides $Interface_c$ to role r is subject to the following condition being satisfied:

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

The role's *expected interface* serves as a *"scope"* for the specification of policies associated with that role. Policies can be written in terms of the functionality specified by the role's expected interfaces 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. This is discussed in Section 3.4.

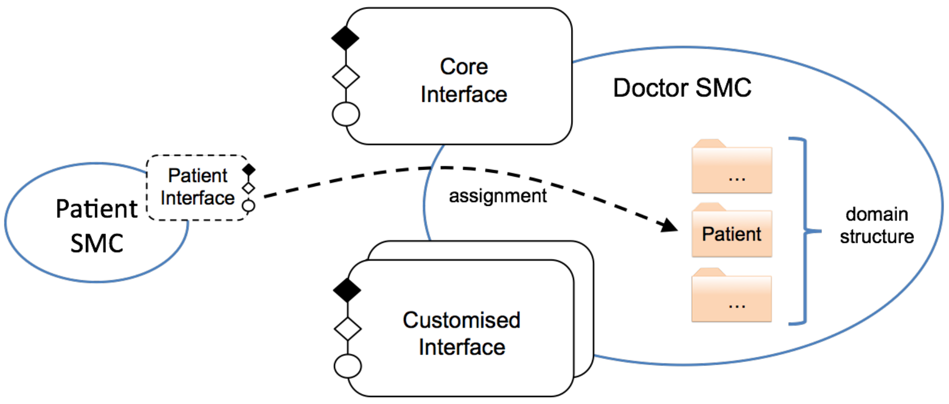


Figure 3.4: SMC assignment

Complex interactions can be encoded in terms of a group of policies that are dynamically loaded and enforced by one or more SMCs within the context of an interaction, as discussed in the next section.

3.4 Missions: Behaviour of SMC Interactions

Complex SMC interactions can be achieved by the exchange of policies in conjunction with the events required for triggering these policies. Policies can be used to prescribe how an SMC should behave in the context of an interaction, i.e. how it should react to both internal events and external notifications by invoking management actions locally or on remote SMCs. Thus the ability of loading policies provides a mechanism for altering the behaviour of remote SMCs at run-time. We rely on the concept of a *mission* to support policy exchanges between SMCs. A mission provides a mechanism for grouping the duties (in terms of the obligation policies) that an SMC must perform, and is specified in terms of two or more collaborating roles.

Missions are normally pre-specified by an application "programmer". We rely on the notion of *expected interfaces* (Section 3.3.2) to 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 loaded and instantiated on the discovered SMC. Mission loading and instantiation are dependent on the existence of authorisation policies allowing one SMC to perform these actions on another (these are discussed in Section 3.5).

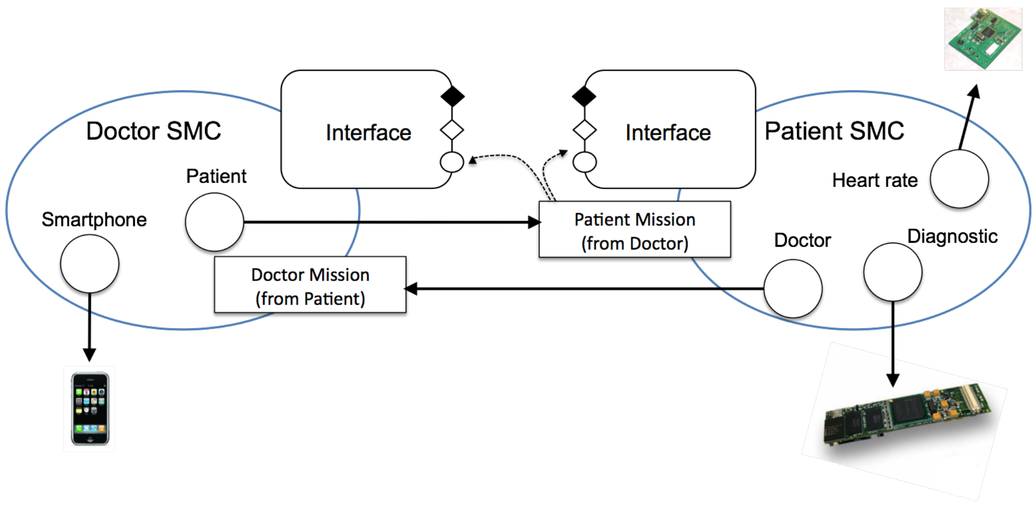


Figure 3.5: SMC mission exchange

Figure 3.5 illustrates a mission exchange between a doctor and a patient SMCs.

When a doctor SMC discovers a patient's body-area network SMC, a mission is loaded 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 fulfill in the interaction, e.g. for re-calibration of the patient's sensors.

3.4.1 Mission Specification

A mission specification is defined in terms of *expected interfaces* of a number of *roles* and consists of a set of *obligation policies* that must be loaded and instantiated in a remote SMC performing role r. Formally, let R be a set of roles, O a set of obligation policies and M a set of missions. A mission specification $m \in M$ is defined as:

$$Mission_m = \langle R_m, O_m \rangle$$

Where:

- $O_m \subseteq O$ is a set of *obligation policies* with the subject role r, i.e. O_m defines the duties of the SMC that will be performing role r; and
- $R_m \subseteq R$ is the union of the target roles in O_m and the role r itself, i.e. R_m are the roles needed for enforcing the obligations contained in the mission.

The mission specification may also specify an array of application-specific arguments, e.g. thresholds, measurement rates, etc. At mission loading, values for these arguments must be provided, which will be used when the policies defined within the mission are instantiated in the target SMC, e.g. raise the alarm if the patient's heart rate is *"rate $\geq v$"*, where v is the value provided at mission loading.

Figure 3.6 shows an example of a mission for *ECG monitoring*, which is typically downloaded from a nurse into a patient SMC. In addition to the *"nurse"*

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

Figure 3.6: Patient monitoring mission (Ponder2 XML notation)

and *"patient"* roles, the mission takes two additional application-specific arguments (*"time"* and *"frequency"*), whose values are specified when the mission is instantiated. This mission defines the obligations that patients must enforce in order to enable a nurse to perform an ECG. The specific nurse and patient SMCs that are expected to participate in this interaction are also given upon instantiation. Policies are specified in terms of the roles and application-specific arguments (*"!rolename;"* and *"!argumentname;"*, respectively).

The mission in Figure 3.6 comprises an obligation policy named *"ECGMon"* which is triggered by a *"startMonitoring"* event received from the nurse. The policy action causes the patient to schedule two tasks: one reads the patient's ECG for a specified time and at a specific frequency, and the other notifies the nurse when the monitoring has finished. This mission relies on the methods *"scheduleTask"* and *"readECG"* that are *expected* to be present in the patient's customised interface, and on events that must be either generated or received by the SMCs. The operation *"notify"* is part of the nurse's core interface. The argument of this method must be one of the notifications defined in the nurse's interface. The operation *"load"*, not shown in the example, is used to load a mission and is also part of the core interface of all SMCs.

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.

3.4.2 Security Concerns in Mobile Code

When considering the ability of dynamically loading executable or interpretable code into remote resources, security concerns must be taken into account. In this section we briefly review the most frequent techniques for ensuring the integrity of the target resource when running remote code: *Proof-Carrying Code (PCC)* and the *Java Sandbox* model. We then elaborate the security requirements that have to be considered when missions are exchanged between SMCs and describe how we address these issues in sections 3.4.3, 3.4.4 and 3.4.5.

Proof-Carrying Code

Proof-Carrying Code (PCC) [NL98] can be used to determine whether the program code provided by a source system is safe to install and execute in a target system, without requiring interpretation or any run-time checking. The notion behind PCC is that the source system attaches to the code a proof that this code does not violate a *safety policy* specified by the target system. This policy, which is specified in first-order logic, defines the safety rules that the target system desires to enforce for the untrusted code, e.g. memory safety, time limits on execution and resource usage bounds. The proof is checked by the target system using an automatic proof-checking process. The main advantage of the PCC approach is that, after the proof has been validated, there is no need for the target system to perform any run-time safety checks.

PCC comprises two main stages: *proof generation* and *proof checking*. In the first stage, the code consumer receives the untrusted code and extracts from it a *safety predicate*. The code must be inspected in search of instructions whose execution violates the *safety policy* defined by the target system. For each instruction, a *predicate* is produced, which expresses the conditions where the execution of the instruction is safe. The combination of these predicates forms a *safety predicate*, which can be proved true only if the execution of the code does not violate the target's *safety policy*. This predicate is then sent to a proof producer (either the source system or any other proof producer system), which must prove the safety predicate using the axioms and inference rules defined in the target's safety policy. In the second stage, the target system receives the proof from the proof producer, and checks the validity of the proof using a simple and fast automatic proof checker, which is parameterised with the proof and the safety predicate, and using the safety policy it determines whether the proof proves the safety predicate or not. The code is considered safe to execute if the safety predicate is correctly proved.

The safety predicate must be relatively easy to prove without extra knowledge about the program to perform proof generation automatically. However, automatic decision procedures do not exist (or are not effective) when generating proofs for more complex safety predicates. In these cases, a semi-interactive theorem prover is required, involving a person with a deeper understanding of the code [NL98].

The Java Sandbox Model

Java supports the ability to load, on demand, programs into a remote resource. The *Java Sandbox* provides the security model for applets and any other Java applications, which restricts the actions performed by a remote program within certain boundaries. This is also required for SMC missions, whose actions that are executed must be confined to those determined by the target SMC.

In this section, Java is used to illustrate the verification steps that must be performed when remote code is loaded into a device. Loading missions should not be mistaken for loading Java bytecodes, which are a much more general form of programming and whose verification is also more complex. Instead, we use the Java model only as an analogy for identifying the various stages involved when remote SMCs load missions into each other.

The enforcement of the Java language rules occurs in three stages [Oak01]:

- *Compiler enforcement:* during development time, compiling the Java source code;
- *Bytecode verification:* at the time the code is loaded into the remote host (before its execution); and
- *Run-time enforcement:* is performed continuously during the execution of a Java program.

The compiler performs various syntactic checks to ensure that the Java program complies with the rules defined in the language. These checks include verifying the access level of attributes, methods and classes (private, default, protected, public), that variables are not used before they are initialised, etc. Bytecode verification is needed because a remote malicious compiler may have by-passed the language rules that were supposed to be checked during the compiler enforcement stage. The main checks include verifying that the classes are in the correct format, that final methods are not overridden, that there is no illegal data conversion of objects, etc. A special case of bytecode verification is called *delayed verification*, which delays some of the checks, provided these are still performed before the code is executed. Finally, run-time enforcement performs checks during program execution, such as array bound checking and object casts.

In particular, one error that may be raised during run-time by the *Java Virtual Machine* is of interest: *java.lang.NoSuchMethodError*. This occurs if an application tries to call a specific method of a class, but the class no longer has the definition of that method. Normally, the compiler would detected this condition, and this error can only occur if the class definition has changed during run-time. Similarly, the interfaces that a mission depends on might be modified during its execution, e.g. one of the sensors provided by the patient may fail, and an SMC must be prepared to handle this situation.

SMC Remote Policy Mode

Proof-Carrying Code and the Java Sandbox model present two distinct ways for ensuring that the code loaded from a source system will not compromise the integrity of the target system. While PCC relies on the validation of a proof that the code does not violate the safety policies in the target system, the Java model confines the execution of the program within certain bounds in the target system, and relies on three different steps where the security rules can be enforced.

We chose to use an approach which is based on the Java model for defining the security aspects of mission exchanges, as the more complex proofs in PCC require a person with deep understanding of the code to define the proof. Additionally, the proof in PCC may be several times longer than the actual code, thus this approach is not suitable for resource-constrained devices. Finally, PCC also requires the translation of the original code (that can be written in any language) to a stream of instructions in a generic *intermediate language* [NL98], which will abstract most of the constructs in the original language that are not relevant to the safety policy.

Similar to the Java security model, an SMC mission must be checked in three stages:

- *Mission specification:* must ensure that the source SMC has written a mission that is syntactically correct and is equivalent to the compiler verification;
- *Mission loading:* allows the target SMC to perform verifications before executing the remote code. Similar to *bytecode* verification the target SMC must check the code it receives from remote SMCs to prevent malicious or accidental compromise of the target's integrity;
- *Mission execution:* run-time verification that checks whether any interface, that affects the behaviour of the mission, has changed during the execution of the mission.

The next sections will elaborate each of these stages.

3.4.3 Specifying a Mission

When a mission is specified, an initial verification is performed to ensure that it complies with the *expected interfaces* of the roles involved. The mission is considered valid if all the operations, events and notifications used by the policies within the mission are also defined in the *expected interface* of the respective roles.

Formally, let $Mission_m = \langle R_m, O_m \rangle$ be a mission, and for all $r \in R_m$, let $I_{r,m}$ be the expected interface of r, then $Mission_m$ complies with the expected interface of its roles if:

$$\begin{aligned}\forall\ obl \in O_m \mid\ & (r.operation \text{ ``}appears\ in\text{''}\ obl \rightarrow r \in R_m \wedge operation \in O_{I_{r,m}}) \\ & \wedge (r.event \text{ ``}appears\ in\text{''}\ obl \rightarrow r \in R_m \wedge event \in E_{I_{r,m}}) \\ & \wedge (r.notification \text{ ``}appears\ in\text{''}\ obl \rightarrow r \in R_m \wedge notification \in N_{I_{r,m}})\end{aligned}$$

Where $\langle a \rangle.\langle b \rangle$ "$appears\ in$" $\langle c \rangle$ means that the operation, event or notification "b" of role "a" is used in the specification of policy "c". This verification is equivalent to a syntactic checking of the mission.

3.4.4 Loading a Mission

When the source SMC loads a mission specification into the target SMC, the source must specify the parameter values to be used upon mission instantiation on the target side. The source defines the SMC instances needed for the mission. The target must then instantiate the obligation policies contained in the mission using the parameter values provided by the source. The target must ensure that the mission passes through a series of checks, when loading the mission and before instantiating its policies, in order to protect the SMC's integrity. These verifications must guarantee that the *mission is well-formed*, that the *mission parameters were provided correctly* and that all *mission dependencies can be satisfied* within the target's local environment. These checks are discussed in the following.

Step 1: Check that the Mission is Well-Formed

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

Step 2: Check Mission Parameters

When a mission provided by a source SMC *A* for a target SMC *B* interacts with a third party SMC *C*, it is necessary to check that *B* is using the correct relevant interfaces for *C*. The source SMC *A* cannot disclose to *B* the *customised interfaces* that it has acquired for interacting with remote SMC *C*, i.e. the argument values for the roles involved in the mission cannot be the customised interfaces for the corresponding SMCs. Instead, the values must specify the address of those SMCs, and it is the target's responsibility to acquire specific customised interfaces for executing the mission.

For example, consider a doctor loading a mission on a patient SMC, which involves three roles: doctor and patient (as these are respectively the source and target of the mission), and an additional nurse role (assuming the mission involves invocations on a nurse interface, e.g. for setting an alarm off). In this case, the doctor SMC cannot simply give an interface for the nurse SMC to the patient, as it is likely that doctors and patients will have access to different customised interfaces provided by nurses. Instead, the doctor must provide the address of the specific nurse SMC, leaving it up to the patient SMC to acquire a suitable *customised interface* for that interaction (through the nurse's *core interface*).

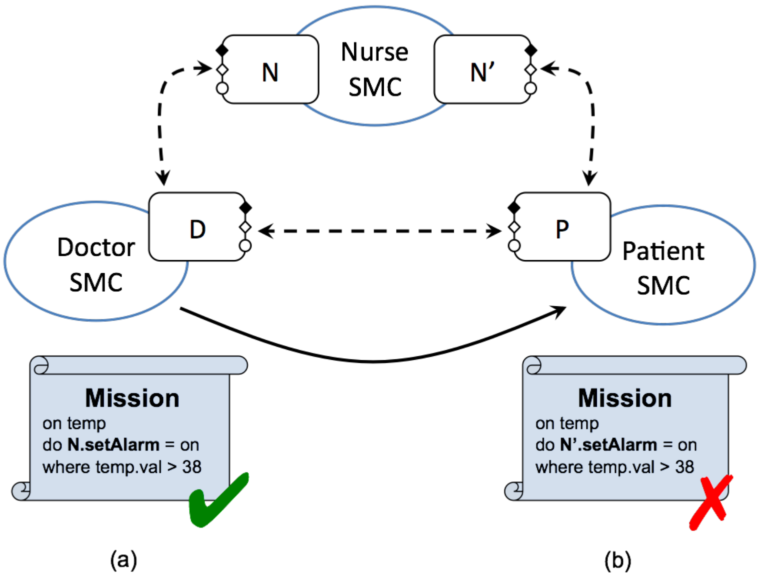


Figure 3.7: Target SMC possesses more restrictive interfaces than source SMC

In Figure 3.7, the dashed lines represent interface exchanges (*D* is for doctor, *P* is for patient and *N* is for nurse) and the solid line represents mission loading. In this case, the nurse SMC exposes different interfaces to the doctor and patient SMCs (*N* and *N'*, respectively). When receiving a mission, the patient SMC must contact the nurse directly to obtain *N'*, which possibly defines a much stricter set of functionality than *N*. This requires an additional check to be performed by the target SMC, which is described in the following.

Step 3: Check Mission Dependencies

When receiving a mission, the target SMC must check the policy dependencies, i.e. *events*, *notifications* and *operations* used by the mission's policies, against the interfaces of the SMCs which the target has access to. This can be achieved by checking the policies within the mission against either: *(a)* the *expected interfaces* the target knows for those SMCs or *(b)* the *customised interfaces* the target has obtained from the remote SMCs once it has established an interaction with them. The former is sufficient because an interaction is established with an SMC only if the customised interface received from that SMC supports a superset of the events, notifications and operations defined in the expected interface for the respective role. This enables delaying the binding to the remote SMC until it is actually required, but is more restrictive since the customised interface may offer additional operations that are not present in the expected interface. The second approach is more permissive as it allows the mission to contain policies that use operations not present in the expected interface but requires establishing an interaction with the remote SMCs when the mission is loaded.

Formally, the verification against the *expected interfaces* is identical to the one described in Section 3.4.3. The verification in the second case uses the *customised interfaces* obtained from the SMCs assigned to the roles, rather than the *expected interface* associated with that role.

3.4.5 Executing a Mission

If all the verifications performed during mission loading are successful, the target SMC is ready to instantiate the obligation policies contained in the mission. These are template policies, which were written in terms of the roles participating in the mission and in terms of other application-specific arguments, e.g. thresholds, measurement rates, etc. Role references are resolved using the *customised interfaces* acquired by the target for the respective SMCs. Other arguments are directly substituted for the values provided by the source SMC. The policies contained in the mission are then instantiated in the target SMC.

Complex policy-based collaborations can be realised if multiple missions are exchanged between the SMCs, e.g. the doctor can load a monitoring mission on the patient, and both the patient and the nurse SMCs can load missions into each other in a similar fashion. The policies loaded in a mission can trigger internal actions within the target SMC based on events occurring in either of the involved SMCs, or trigger remote invocations on the other SMCs as well.

During mission execution, the interface of the involved SMCs may change, for example, if an SMC acquires new resources or loses current ones, causing the functionality that its interface provides to change accordingly. This can affect mission execution, as the functionality that the mission expects (checked on mission loading) may not be available any more.

When the interface of an SMC is modified because a resource has left, this SMC automatically notifies its interacting partners using the event *leftSMC*, which triggers partner SMCs running the mission to re-check the mission dependencies with respect to that specific interface. This process is akin to the check performed in *step 3* of the mission loading verification. If the SMC determines that the modified interface no longer satisfies the mission dependencies, the mission is stopped and the source of the mission, which originally loaded it into the target SMC, is notified that the mission execution was interrupted.

3.5 Access Control for SMC Interactions

An SMC must be authorised in order to perform operations on a remote SMC. Access control in SMC interactions is defined in terms of *authorisation policies*[1] relating to operations in both *core* and *customised* interfaces.

Permissions are required for invoking application-specific operations on a *customised* interface, either directly or as the action prescribed by an obligation policy. There is a one-to-one correspondence between what is exposed in a customised interface and the authorisations required for invoking application-specific operations on that interface. For example, if a doctor SMC is willing to expose a given functionality to a patient SMC through a customised interface, the doctor must set the corresponding authorisations to allow the patient SMC to invoke the operations on that interface. At the moment, we require customised interfaces and the corresponding authorisations to be generated manually. In particular, authorisations are finer-grained since the access to a given operation may be conditional on the context of the SMC enforcing the

[1]The access control framework for the SMC and details on how authorisation policies are implemented in Ponder2 are described in [RDD07].

policy, e.g. a medication prescription can be requested only between 9am and 5pm. Intuitively the operations in a customised interface could be automatically generated from the set of authorisations defined for an SMC, but further work is needed to investigate this.

Permissions are also required for invoking operations on a *core* interface, e.g. loading and unloading missions. In this case, the definition of which SMC is supposed to *send* and which SMC is supposed to *receive* missions often depends on the nature of the interaction, e.g. doctors will typically be allowed to load missions into a patient. In Chapter 4 we will discuss the use of *patterns of interactions*. Patterns assist in the specification of *manager/managed* relationships between SMCs, and in deploying the required authorisations for mission loading in the context of an interaction.

3.6 Case-Study: Illustrating Security Management

Collaborations between SMCs are created for a specific purpose: for example, for defining the interactions between doctor, nurse and patient SMCs, or for assembling a monitoring set-up involving the equipment and various resources available at the home setting of a patient; another example is a set of *unmanned autonomous vehicles* (UAVs) or robots assembled in a team for the reconnaissance of hazardous areas or to realise rescue operations after floods or earthquakes, carrying out tasks that are too dangerous for humans to perform.

These cross-SMC interactions concern both the *management of the application-specific aspects*, e.g. healthcare monitoring, as well as the *management aspects of how SMC interactions* themselves are established. In this section we concentrate on the latter and present a case-study illustrating how security management of SMC interactions can be achieved through the use of roles, missions and policies. These interactions can become compromised if malicious SMCs are allowed to join, so there is an interest in the mechanisms which are required for the secure establishment and operation of these collaborations.

The purpose of this section is not to introduce new algorithms or protocols for security management, but instead to illustrate how SMC interactions can be extended easily to include new security management mechanisms as required.

3.6.1 Management Requirements

The mechanisms considered essential for the construction of SMC interactions include *authentication*, *membership management* and *access control*. These are necessary because they guarantee that new members are authenticated before being allowed to join the interactions, that failures of current participants can be promptly detected, and that access control is applied to the services provided by the participants of an interaction. Additionally, support for bootstrapping and task-allocation within an interaction is required via a set of *coordination* mechanisms. This is essential for enforcing maintenance actions and guaranteeing the integrity of the interactions.

We describe in this section how these management requirements can be expressed as *roles*, which are assigned to different SMCs within an interaction. Distributed SMCs assigned to different roles then cater for the security management of an interaction, by enforcing the policies contained in a mission specified for that role.

- *Coordination role*: is responsible for *bootstrapping* an interaction and *assigning* discovered nodes to roles. Assignment of SMCs to roles is flexibly defined by policies which take into account the discovered SMC's *type* and its *capabilities*, obtained from the *identity message* sent by the remote SMC when discovered. For example, in an SMC interaction for reconnaissance operations, SMCs providing *"video"* capability are more suitable for a *"surveillance"* role, whereas SMCs providing *"storage"* capability are more suitable for an *"aggregation"* role. However, this is still subject to the SMC providing an interface which supports the functionality required for that role's *expected interface*. The SMC in charge of the *coordinator* role loads the *missions* into the respective SMCs according to their roles. The coordinator also loads *missions* for the SMCs assigned to the management roles. These missions contain policies for authentication of other members and for membership management, for example. The implementation of these policies is described in Chapter 6.

- *Authentication role*: is needed to validate the identity of SMCs that want to join an interaction. It relies on the exchange of *public-key digital certificates*. The SMC assigned to the *authenticator* role is initially loaded with the public-keys of the certification authorities (CAs) that are relevant within the context of an interaction, e.g. *British Medical Association (BMA)* certificates in an application for healthcare monitoring. Nodes that wish to join the interaction must present certificates signed by these authorities to the authenticator. However, this simple mechanism does not cater for key revocations, and the use of non-PKI based authentication [KZ03, SBS$^+$02, SA00] would need to be investigated.

- *Membership management role*: builds upon the functionality provided by the SMC's discovery service to monitor the presence of member SMCs. This is required because nodes may move out of communication range, run out of battery power or disconnect. Participants are required to renew their membership periodically with the SMC assigned to the *membership manager* role. Membership renewal does not require digital certificates to be revalidated, and renewals consist of simple events sent to inform that a given SMC is still active in the context of the interaction. The membership manager keeps track of membership renewals that are received from other members. Each time a renewal from a given SMC is received, the membership manager extends the validity of that member's entry for a limited amount of time. Whenever the membership manager detects that a member's entry has expired because the member failed to renew its membership within a given time frame, that SMC is considered to have left and an event is propagated to the other participants of the interaction.

- *Access control*: is necessary for protecting the resources and services provided by each participant from unauthorised access. Its enforcement is distributed amongst all roles, in the form of authorisation policies, as typically each SMC is interested in protecting its own resources and permit access to specific members within the interaction. However, if an SMC is not capable of enforcing its own access control policies, it may outsource these control decisions to another SMC or to its own trusted agent.

Specific SMC interactions can be extended with additional management roles if necessary. The SMC infrastructure is not specific to a particular security model. For example, *threshold cryptography* [ZH99] for preventing a compromised authenticator from accepting rogue members in the interaction and *intrusion detection* [Sun96, Lun93] for monitoring potential risks and attacks are examples of the mechanisms that can be added to a given interaction. Role allocation strategies such as the one in OASIS [BMY02, YMB01, HYBM00] could be encoded as a pattern for role assignment and mission loading. OASIS uses a formal logic to specify precise conditions for entering a role, which are based on the node's credentials. The node is then issued with a *role membership certificate (RMC)* that can be used subsequently as one of the node's own capabilities to satisfy the conditions for entering a further role (a proof tree is built which corresponds to the dependencies between these certificates). The much simpler role assignment based on capabilities, as described earlier, was used to illustrate the management aspects of SMC interactions. Our objective however is not in developing such mechanisms but rather to illustrate how security and management procedures can be enforced amongst distributed participants of an interaction according to their roles.

3.6.2 Management of SMC Interactions

Initially, the SMC which is assigned to the coordinator role may be also assigned to other management roles (e.g. authenticator role), since it alone initiates the interaction. However, as new SMCs are discovered, these functions can be devolved to them by assigning new SMCs to the roles and loading the respective missions. Figure 3.8 succinctly describes the overall operation of an

SMC interaction. The SMC assigned to the coordinator role bootstraps the interaction and periodically broadcasts its *identity message*, which also includes in this example the address of the authenticator *(1)*. An SMC that is interested in joining the interaction contacts the authenticator and sends its credentials. This in turn triggers a policy within the authenticator, which causes the credentials of the discovered SMC to be verified using the public-keys previously loaded in the authenticator *(2)*. Our implementation uses authentication based on *X.509 digital certificates*, but other strategies could be loaded into the authenticator in the form of policies.

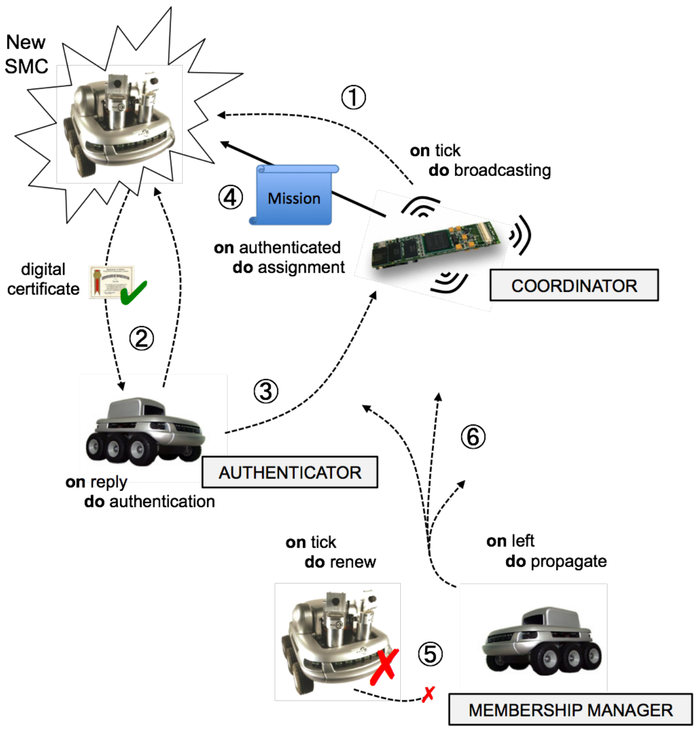


Figure 3.8: Management and simplified policy-based interaction

If the credentials of the discovered SMC are successfully validated (and if needed, the authenticator's credentials are also successfully validated by the new SMC), the authenticator sends an event to the coordinator, informing it that the discovered SMC is allowed to join the interaction and what the SMC's capabilities are *(3)*. Policies in the coordinator specify the preferable role assignment strategy matching the role requirements and the SMC's capabilities. The role assignment process includes transferring the *missions* and authorisation policies that are meant to be enforced by that specific role to the new SMC *(4)*. A mission specification $m = \langle R, O \rangle$ groups the obligations O that correspond to the duties of the SMC and defines the set of roles R which are needed for the enforcement of the policies. The SMC also receives the identity of the members assigned to each role, and then directly contacts them to acquire the respective *customised interfaces* for each one of these SMCs (as described in Section 3.4).

Membership management is also policy-based. Policies require each participant to send an event periodically renewing its membership with the membership manager *(5)*. In turn, the membership manager monitors the presence of other members. The renewal event received from a member triggers a policy in the membership manager which extends that member's entry for a given amount of time. When an entry expires because it has not been validated, an event is raised locally in the membership manager, informing that the SMC has left the interaction. This causes the membership manager to notify all other participants so they can check whether they are still capable of enforcing their missions *(6)*. There is an obvious trade-off between how often SMCs should revalidate their entry and how accurate the membership lists should be kept. For this reason, these actions were defined as policies which can be easily changed to adapt updating rates (and thus accuracy) to different requirements.

3.7 Discussion

This chapter identified basic mechanisms for supporting SMC interactions. Through its interface, an SMC specifies which operations it provides for remote SMCs, and which events the SMC is able to send or to receive from remote SMCs. Whilst the functionality provided by the *core interface* is application-independent and common to all SMCs, the *customised interfaces* of an SMC specify different sets of application functionality that are offered to their interacting partners. When a remote SMC is discovered, it is assigned in the discoverer's domain structure, in a role that is compatible with that SMC's interface. Operations can be invoked and events can be exchanged, but more complex interactions will typically involve policy exchanges, allowing an SMC to load a set of obligations into a partner SMC, in the form of a *mission*. Missions will only be accepted by a target SMC if it can verify the mission's integrity, as missions are equivalent to a constrained form of programming remote SMCs. Invocation of operations and mission loading are subject to authorisation policies allowing one SMC to perform these actions on a remote SMC.

Cross-SMC interactions concern both the *management of the application-specific aspects*, e.g. healthcare monitoring, as well as the *management aspects of how SMC interactions* themselves are established. In this chapter we presented a case-study illustrating how security management of SMC interactions is achieved through the use of the basic SMC elements, such as policies, events, interfaces and roles. The purpose of this case-study was not to introduce new algorithms or protocols for security management, but to illustrate how the same principles apply to the management of the interactions themselves.

Interactions between SMCs are encoded in terms of policies and events, according to the role each member performs. However, to apply this policy-based infrastructure to the management of large-scale systems, it is necessary to be able to structure complex SMC interactions, involving specific task-allocation and event-forwarding strategies that can be rapidly instantiated among groups of SMCs. For example, SMC interactions may be hierarchically composed of smaller interactions, such as a large rescue team which has a medical team as one of its members, that is itself a complex interaction. In this case, smaller interactions should encapsulate their management details in order to allow the system to scale up. Similarly, cross-SMC interactions can be established between a patient body-area SMC involving several other devices and the equipment provided in the GP surgery, as well as between these and the devices available in a home monitoring set-up for example.

The next chapter will introduce the use of software patterns as a means of engineering larger policy-based interactions, or interactions formed by smaller SMC collaborations. Interactions based on patterns rely on the principles presented in this chapter and allow the systematic specification and instantiation of parts of a collaboration by reusing and composing building block abstractions.

Chapter 4

Patterns for Building SMC Interactions

Frequently a collaboration of SMCs may be part of a larger, and more complex interaction. Building elaborate applications using solely elementary abstractions such as policies, roles and events can be difficult to manage and deploy. This chapter describes the use of software *patterns* for engineering and structuring complex policy-based interactions between SMCs. The use of patterns provides a much more general model for the specification and instantiation of SMC interactions and can accommodate a potentially unbounded spectrum of interactions. Patterns allow the specification of SMC interactions through abstractions for their structure, management and communication aspects. Patterns can be reused and applied recursively for composing and federating SMCs in a systematic manner, thus providing support for the construction of large policy-based SMC collaborations.

4.1 Architectural Principles and Motivation

Christopher Alexander's work, *The Timeless Way of Building* [Ale79], has influenced the development of software engineering, in particular through *software* *patterns* and in the use of *architectural styles* for organising and structuring software systems. His work was aimed at the use of patterns for urban planning and building architectural elements in the physical world, and how these can be composed to address complex architectural needs and constraints, but the same principles can be similarly applied for defining the architecture of software elements.

Software architectures and architectural styles as discussed in Chapter 2 typically orchestrate the use of components and connectors as a means of structuring software development [GS93, SDK$^+$95, SDZ96, MMP00, MT00, TMD09]. Although these do not cater for the adaptive behaviour of SMCs, similar principles can be applied for structuring and reusing SMC interactions to form larger collaborations. In this chapter we describe the use of *architectural styles* to assist in the design of policy-based SMC interactions. An architectural style enforces important architectural decisions with respect to the organisation of SMCs, and implements specific algorithms or protocols that constrain how SMCs achieve the required exchanges of policies, events and interfaces. Building SMC interactions solely in terms of these simple exchanges is challenging, and laborious to manage and deploy. In contrast, architectural styles provide a more systematic manner for engineering large policy-based interactions, and support to reason about how SMCs are composed and verify the correctness and the properties achieved by a specific interaction. Architectural styles can express designs and capture solutions in a manner that promotes their reuse across applications. Although the identification and specification of these styles does require human involvement, this is unavoidable because frequently best design practices tend to be domain-specific and dependent on experience. Architectural styles are similar in intent to software design patterns [GHJV95] in the sense that they provide a set of standard solutions for recurring problems.

The consistent exchange of policies, events and interfaces can be seen as distinct *perspectives* of a policy-based SMC interaction. These perspectives are complementary, as the exchange of policies must be accompanied by adequate exchanges of events (required for triggering the policies) and interfaces (required for validating remote invocations prescribed by a policy). These multiple perspectives can then be used to emphasise and better understand independent aspects of a policy-based SMC interaction, similar to the way that the design aspects of a building can be considered independently (electrical wiring, plumbing and heating, for example) [PW92]. This allows particular aspects of an architecture to be highlighted whilst omitting others [TMD09]. We are interested in abstractions that concern three main aspects of SMC interactions:

- *Management*: defines how policies are exchanged between SMCs and under which conditions these exchanges happen. Management (or task-allocation) is achieved through the loading of policies which are grouped into *missions* (see Section 3.4). However, policy exchanges may rely on very specific abstractions. Some collaborations for example are reliant on a single manager and multiple managed SMCs, while others allow multiple managers to load tasks into a single SMC (which requires checking for and resolving conflicts between the loaded policies [LS99]). Task loading can be uni-directional or bi-directional (in the latter, each SMC is both a manager and a managed node). Alternatively, tasks could be loaded according to a bidding strategy where SMCs express their willingness to receive tasks from an issuer. Finally, task-loading can be conditional to a set of criteria, for example based on the capabilities provided by an SMC, its profile or the credentials that the SMC possesses.
- *Communication*: defines flows of information typically through asynchronous events between SMCs, as events are required for triggering policies, although we do not require that all interactions between SMCs be event driven. This varies from a simple diffusion of events from a source to a target SMC to a more elaborate shared-bus between a set of SMCs that works as a blackboard [EHRLR80] for shared events. Additional aggregation functions such as correlation of events provide flexibility in defining event patterns for triggering higher-level events. Alternatively, store-and-forward primitives are useful in ad-hoc settings where SMCs do not have a permanent connection to their partners, and where events must be locally stored for subsequent delivery.

- *Structural*: structural aspects reflect how SMCs are organised with respect to interface access, visibility and encapsulation, as interfaces define the actions used in the policies. For example, peer-to-peer interactions typically rely only on a simple exchange of interfaces whereas compositions also need to implement encapsulation and mediate access to internal resources. Composed interactions allow one SMC to restrict the visibility of its inner resources to external SMCs, creating an encapsulated structure. Additional abstractions such as filtering of operations provide more flexibility with respect to interface exchanges, allowing one SMC some degree of control on the access to another SMC's interface. A particular combination of structural abstractions provides very specific properties for the interface exchange aspects of a collaboration.

The investigation of a number of application scenarios of collaborating SMCs led to the identification of a *catalogue* of architectural styles for SMC interactions, which is presented in the next section. A model for composing and federating SMCs by methodically combining individual architectural styles is then described, which facilitates the building of complex interactions by reusing styles as building block abstractions.

4.2 Catalogue of Architectural Styles for SMC Interactions

A catalogue of architectural styles provides several useful abstractions for defining management relationships across self-managed cells. Our intention is to provide a better understanding of these relationships and promote the reuse of common abstractions for systematically building large-scale policy-based collaborations. Although a large number of different interactions can be defined, typically applications tend to use small subsets of these interactions.

Table 4.1 presents a brief overview of a catalogue of architectural styles for SMC interactions. As in all such catalogues we cannot aim to be exhaustive

Table 4.1: Catalogue of architectural styles for SMC interactions

Category	*Architectural Style*	*Description*
Structural	Peer-to-Peer	Ordinary, symmetrical mode of interaction between SMCs that exchange interfaces
	Composition	One SMC encapsulates another's interface and determines its visibility through mediation
	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
Management	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	Diffusion	Provides a way of directly forwarding events to interacting SMCs
	Shared-Bus	Provides a blackboard for decoupled event-based communication among SMCs
	Correlation	Individual events are combined for generating a higher-level event
	Store-and-Forward	Useful in ad-hoc settings where SMCs do not have a permanent connection to their partners

but focus solely on the frequently occurring styles that facilitate the design and composition of SMCs by structuring the devolution of management responsibilities and corresponding interactions. Each architectural style relies on different abstractions for interface, policy or event exchanges, and correspond to the structural, management and communication aspects of an interaction respectively. These categories can be seen as complementary perspectives for defining policy-based interactions between autonomous SMCs. They are complementary because a given exchange of policies must be accompanied by the adequate exchanges of events and interfaces required for validating remote invocations prescribed by a policy.

In the following we outline individually each one of these architectural styles and Section 4.3 describes how complex collaborations between SMCs can be specified and instantiated by combining architectural styles as design elements of an interaction. To simplify the presentation, the diagrams illustrate interactions between simple SMCs. However, each one of these SMCs may be a complex SMC itself which might have an internal structure comprising several other SMCs.

4.2.1 Structural Styles

Structural styles define how SMCs are organised with respect to the access of their interfaces. These involve abstractions such as encapsulation, or filtering and mapping of interfaces, and also combining two or more constituent SMCs to form a new SMC.

Peer-to-Peer (Figure 4.1) defines a relationship between "equal" peer SMCs, which provide or request services to or from each other, while each peer retains its autonomy and is free to establish additional P2P interactions with other SMCs. No specific semantics of ownership, hierarchical organisation or encapsulation is applicable. Invocations are subject to authorisation policies allowing the partner to perform the actions but no predefined management or control relationships are implied. A peer-to-peer interaction can be established between a set of UAVs in the same rescue mission which collaborate and offer services to each other: e.g. one UAV can request updates about hazardous chemicals detected by a surveyor UAV, while the latter can also use the storage service provided by a third UAV to back-up this information during field operation.

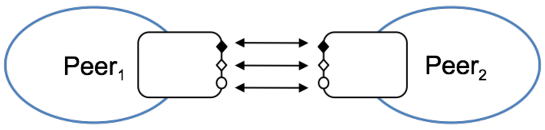


Figure 4.1: Peer-to-peer architectural style

Composition (Figure 4.2) specifies an interaction between SMCs in which an *outer* SMC *encapsulates* one or more *inner* SMCs. The inner SMCs then become managed resources of the outer SMC and the inners' visibility outside the composed structure is subject to and mediated by the outer SMC. The outer may expose parts of the inner interfaces within its own interface. An inner SMC is not allowed to be contained by other SMCs or to directly interact with other SMCs outside the composition. Typically a patient body-area network SMC, running on a Gumstix or smartphone, forms a composition with a set of BSN sensors for health and environmental monitoring which are encapsulated in the body-area SMC.

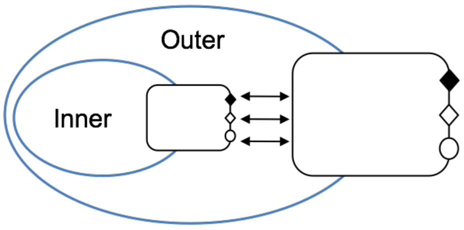


Figure 4.2: Composition architectural style (encapsulation)

Aggregation (Figure 4.3) specifies a relationship in which SMCs form a hierarchical structure and one of the SMCs (*inner*) becomes part of another (*outer*), and may be mediated by the outer's interface. This can be used to form hierarchies of SMCs where lower-level SMCs provide services to higher-level ones, without imposing strict encapsulation among SMCs. Thus in an aggregation the inner SMC can also interact directly with other SMCs outside the aggregation if necessary (as a shared resource). For example, a specific UAV which is part of a team of UAVs can be loaned to cooperating teams if required, which then can directly access the shared UAV.

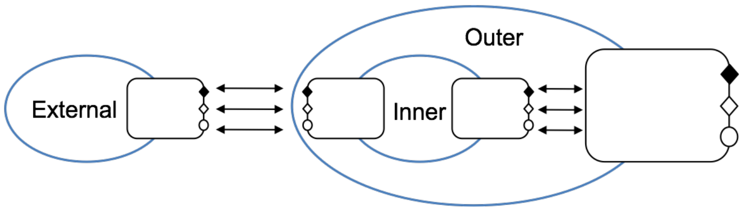


Figure 4.3: Aggregation architectural style (no encapsulation)

Fusion (Figure 4.4) combines two or more constituent SMCs into a new SMC, and the forming SMCs cease to exist. The resulting SMC provides an interface combining the functionality previously provided by each constituent interface, and enforces the set of policies previously enforced by each constituent SMC. Finally, this new SMC must take over the collection of managed resources possessed by each constituent SMC. Fusion is more natural among complex SMCs that represent logical collections of elementary SMCs. As an example, a fusion may occur between two SMCs representing teams of UAVs on related missions, which are combined to form a single larger team with a single commander for all the UAVs.

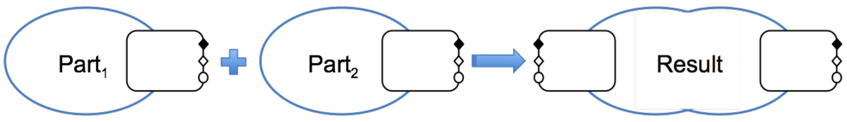


Figure 4.4: Fusion architectural style

4.2.2 Management Styles

Management styles capture task-allocation strategies and control aspects. The tasks exchanged between SMCs are specified in the form of *missions*, as discussed in Section 3.4. Missions are groups of policies that can be dynamically loaded to change the behaviour of interacting SMCs at run-time. While missions define *what* tasks are being exchanged, management styles specify *how* these exchanges occur. Management styles also assist in the specification of *manager/managed* relationships between SMCs, facilitating the deployment of the authorisation policies which are required for mission loading. Management styles are complementary to the structure of an interaction, and architectural styles of both categories can be combined. For example, a group of SMCs establish a *peer-to-peer* structural interaction (with no encapsulation or mapping in the access of SMC's interfaces) and use a distribution of tasks based on an *auction* style. In other cases the same *auction* style can be used to define the task-allocation, while relying on a *composition* for the structural organisation of SMCs.

Hierarchical Control (Figure 4.5) consists of a top-level SMC which controls the execution of a set of leaf SMCs by delegating policies to them. This implies that the top-level SMC has rights of programmability over the leaf SMCs, i.e. the top-level SMC is authorised to load tasks and policies into the leaf SMCs. It is a unidirectional interaction in that only the top-level SMC loads tasks into the leaf SMCs. For example, in a search-and-rescue team a commander UAV may have rights to manage the task-allocation of the various subordinate UAVs. This does not imply that access to leaf-SMCs is mediated by the top-level SMC, as the hierarchical control style is only concerned with task-allocation. Hierarchies can obviously be multi-level.

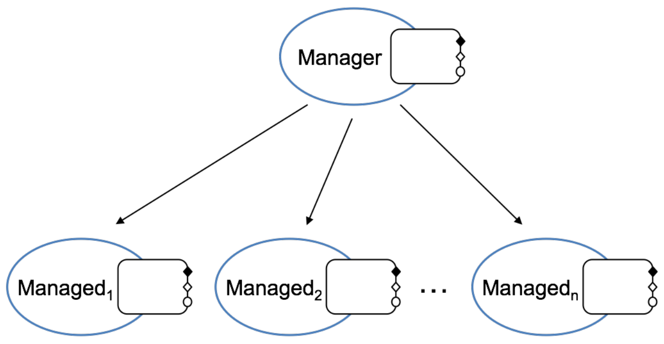


Figure 4.5: Hierarchical control architectural style

Cooperative Control (Figure 4.6) defines a set of cooperating top-level SMCs that control the execution of a leaf SMC by delegating policies to it. This implies that the top-level SMCs have rights of programmability over the leaf SMC. Similar to the hierarchical control style, this is a unidirectional interaction in that only the top-level SMCs load tasks into leaf SMCs. These multiple managers may cause conflicting policies to be loaded into the leaf, and this style must address mechanisms to solve such conflicts, e.g. prioritisation of policies or use of application-specific meta-policies [LS99]. For example, a shared UAV subject to policies loaded from two cooperating coalitions, e.g. a US and a UK coalition, relies on the cooperative control style to address the policy conflict issues that may arise from this interaction.

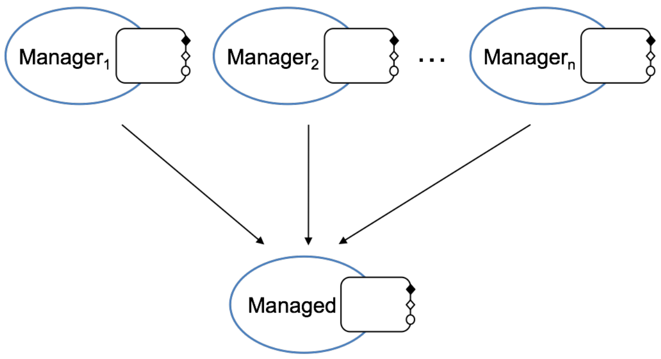


Figure 4.6: Cooperative control architectural style

Auction (Figure 4.7) provides a task-allocation model based on the *contract net* approach used in multi-agent systems [Smi88]. It facilitates task distribution using negotiation, where tasks are announced and SMCs decide whether to bid for their execution. The top-level SMC (*issuer*) evaluates the bids and assigns the task to the most appropriate leaf SMCs (*bidders*). The issuer SMC decides what tasks to offer and bidders decide which tasks to bid for, possibly after a policy defined negotiation. The decision of which bids to accept is made by the issuer, who cannot directly impose task execution on a bidder. Thus, the auction style does not imply that the top-level SMC has rights over the leaf SMCs and authorisations are only granted when the negotiation is successfully completed. The negotiation process is application-dependent, but typically involves attributes related to the leaf nodes, e.g. knowledge bases available, capabilities, current workload, etc. It is a unidirectional interaction in that only the

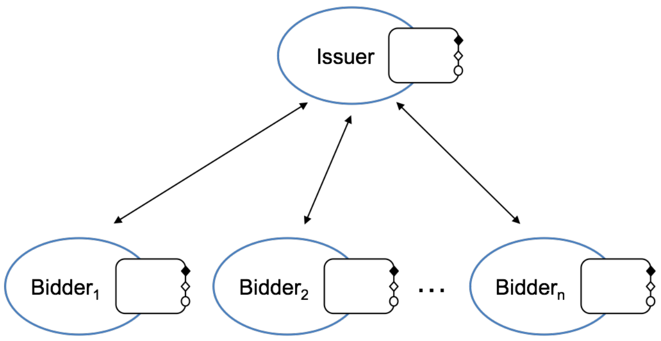


Figure 4.7: Auction architectural style

top-level SMC is able to load tasks into the leaf SMCs. Task-allocation based on capabilities available in autonomous robots [CKC04, INPS03] could be used to encode bidding protocols for SMCs.

Distributed Control (Figure 4.8) represents a fully decentralised task-allocation interaction among a set of SMCs. The nature of such an interaction is application-dependent, where both SMCs act as managers and managed resources simultaneously and have rights of programmability, i.e. authorisations, over each other. Policies associated with this style address general authorisations amongst the SMCs, and general conflict resolution rules similar to those provided by the cooperative management style. For example, a doctor SMC may load health-monitoring tasks into a patient SMC. Similarly, a patient SMC may also need to load policies onto the doctor SMC to trigger re-calibration of the patient's sensors if needed, as this avoids the requirement for the doctor device to store calibration procedures for all possible patients. In this interaction both SMCs have the authorisations required to load tasks into the respective partner.

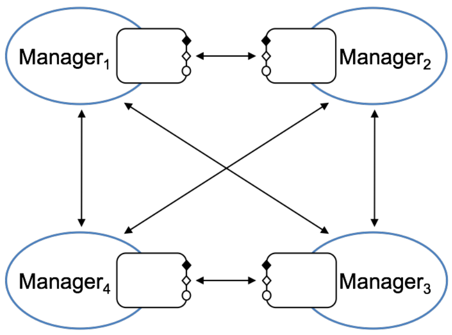
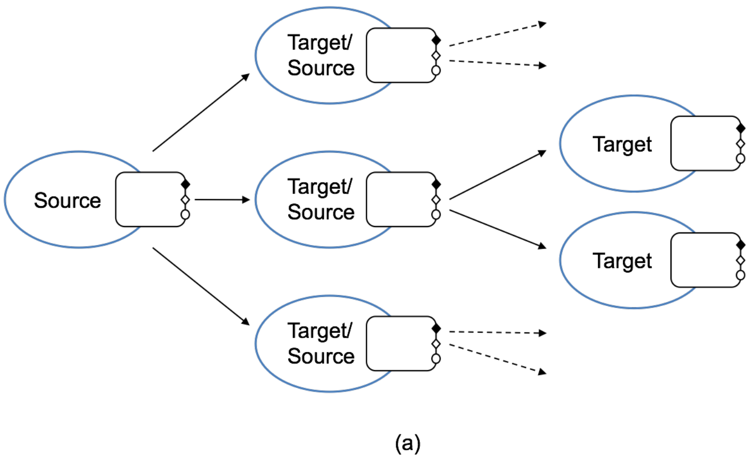


Figure 4.8: Distributed control architectural style

4.2.3 Communication Styles

Communication between SMCs typically occurs through asynchronous event exchanges indicating context changes within an SMC, as events are required for triggering policies running on remote SMCs. This category of architectural styles defines patterns that specify how events are exchanged and how the event buses of various SMCs are interconnected. The intent of these styles is not to restrict the ways SMCs can communicate, but instead to provide common means of event-forwarding patterns that will facilitate the implementation of specific interactions.

Diffusion (Figure 4.9) defines the forwarding of events from a *source* to a *target* SMC. Diffusion has been widely used in computer network protocols and data dissemination algorithms [Tan96]. This style can be expressed in different topologies: Figure 4.9*(a)* shows the general case of *branching diffusion*, while Figure 4.9*(b)* shows an example of a sub-case known as *linear diffusion* or *pipeline*. This style specifies rules regarding the forwarding strategy and routing protocols to be employed. Ad-hoc sensors monitoring environmental data in the field are likely to use the diffusion style to propagate monitored data towards processing and logging "sinks".



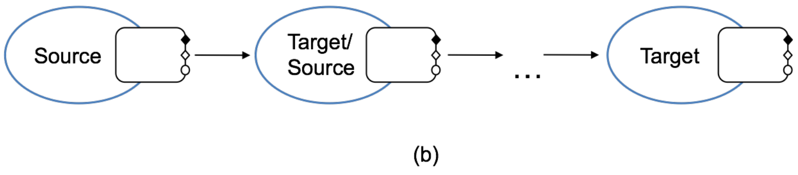


Figure 4.9: Diffusion architectural style: branching diffusion (a), which is the general case, and linear diffusion (b), which is similar to a pipeline

Shared Bus (Figure 4.10) allows SMCs to interconnect their event buses to a central SMC, which then relays published events to all connected SMCs. SMCs can use the shared event bus as a blackboard for decoupled interactions, be-

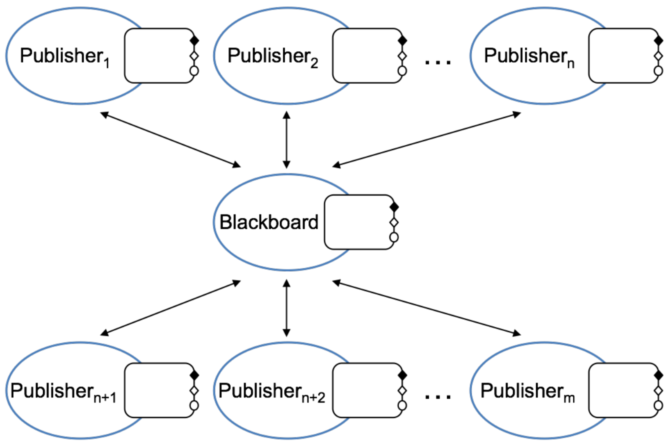


Figure 4.10: Shared bus architectural style

cause the publisher SMC does not require prior knowledge of the recipient SMCs when publishing an event. Only selected types of events may be published in the shared event bus, and an SMC has the choice of publishing certain events only locally in its own event bus. This style can be used for communication among surveyor UAVs which collect layout information of a specific area, where data about new obstacles found by a UAV is published in the shared bus which permits all surveyors to keep a consistent view of the environment.

Correlation (Figure 4.11) addresses event correlation between SMCs. It can be used for collecting events from different sources and generating higher-level events or for collecting patterns of events over a period of time and generating

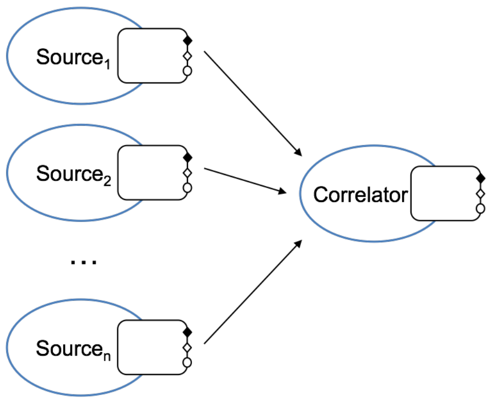


Figure 4.11: Correlation architectural style

another event containing synthesised information, rather than just forwarding raw events. For example, the information collected from various physiological sensors in a body-area network can be correlated in a coarser-grained event, such as a periodical report on the patient's health condition.

Store-and-Forward (Figure 4.12) addresses issues on how to transfer events between SMCs using the principles of delay-tolerant networking [Fal03]. It is useful in ad-hoc settings where an SMC does not have a permanent connection to all other SMCs it wishes to interact with. The store-and-forward style enables SMCs to communicate via multi-hop interactions, where SMCs act as carriers of events targeted to other SMCs. This style specifies how to store and transfer events to neighbour SMCs according to their availability, ensuring that undelivered events are retained in the SMC for later delivery.

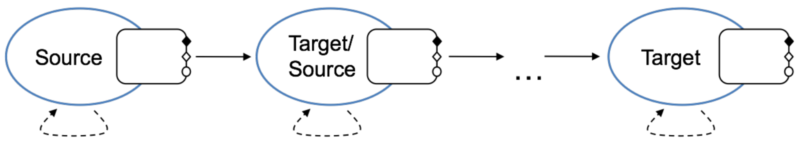


Figure 4.12: Store-and-forward architectural style

4.3 Composing and Federating SMCs

Based on the catalogue of architectural styles we present in this section a methodology for engineering policy-based SMC interactions, which relies on the combination of architectural styles as design elements of a collaboration between SMCs. Under this perspective, each architectural style is used for describing a specific abstraction for either:

(a) the exchange of policies; or

(b) the exchange of events required for triggering policies; or

(c) the exchange of interfaces for validating the actions prescribed by policies.

As described in Chapter 3, SMC interactions rely on the notion of *roles* as placeholders for remote SMCs discovered at run-time. Roles are kept in the *domain structure* that implements a hierarchical namespace within each SMC. A remote SMC is assigned to a role in another SMC if the former fulfills the requirements for that role, e.g. credentials, capabilities. When an SMC is assigned to a role, policies specified for that role will automatically apply and the respective missions will be loaded into the SMC.

In order to support the systematic specification and instantiation of complex SMC interactions, we present a model which permits entire *architectural styles* to be associated with (*bound to*) a group of *roles*. Each architectural style defines a particular algorithm or protocol governing how the SMCs that will be assigned to these roles should behave with respect to interface, event or policy exchanges. When SMCs are assigned to roles, the architectural styles associated with their roles are *instantiated*[1] and SMCs will establish an interaction which is dictated by these styles.

The use of architectural styles provides a better understanding of the relationships between SMCs and promotes reuse of common abstractions. The next chapter will describe how the use of styles also facilitates the analysis and verification of SMC interactions.

4.3.1 Bindings and Interaction Specification

We distinguish between specification and instantiation of an interaction. The *specification* consists of a number of architectural styles which are associated with specific roles in the local domain of an SMC, e.g. *doctor*, *patient*, *sensor*. This defines how SMCs that will be eventually assigned to these roles are expected to behave towards each other. When actual SMCs are assigned to roles, the styles which were previously bound to these roles will be *instantiated*, and interactions will occur in the form of exchanges of policies, events and interfaces as prescribed by the styles.

[1]For the purposes of this discussion, the terms *instantiation* and *deployment* of an architectural style are used as synonyms.

Each architectural style defines its own set of *style-specific roles*, which refer to the abstraction being enforced. For example, a *composition* style defines the roles *outer* and *inner*, whereas a *hierarchical control* defines the roles *manager* and *managed* and a *diffusion* style defines the roles *source* and *target*.

Roles in the local domain of an SMC are *bound* to the style-specific roles of a particular architectural style. These *bindings* specify how remote SMCs that will be assigned to those roles should behave within the context of an interaction, e.g. as an *inner* w.r.t. a *composition*, as a *source* w.r.t. a *diffusion*.

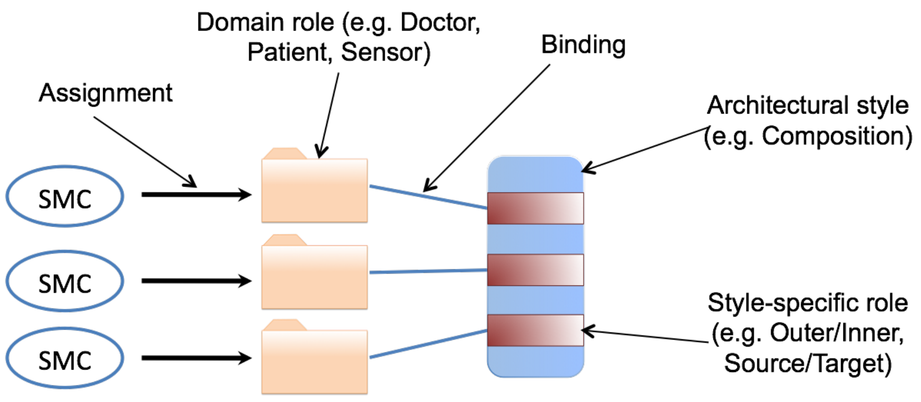

Domain role (e.g. Doctor, Patient, Sensor)
Binding
Assignment
Architectural style (e.g. Composition)
SMC
SMC
SMC
Style-specific role (e.g. Outer/Inner, Source/Target)

Figure 4.13: Architectural styles and bindings

Figure 4.13 illustrates how roles in the domain within an SMC, architectural styles and style-specific roles interrelate with each other: a given architectural style, e.g. *composition*, is chosen and style-specific roles of this style, e.g. *outer* and *inner*, are *bound* to roles, e.g. *doctor*, *patient*, *sensor*, defined within an SMC's domain. This corresponds to the specification of an interaction between a group of roles. When remote SMCs are discovered and *assigned* to these roles, such a style will be instantiated, thereby effecting a *behaviour* among the respective SMCs. This *behaviour* is dependent on the semantics of the style itself, as it refers to how interfaces, policies and events are exchanged.

An architectural style also specifies a set of *requirements* that roles have to fulfill (or rather the SMCs assigned to these roles have to fulfill). For example, a style that specifies the forwarding of the event *highHR* from one SMC to another will typically require this event to be supported by the *source*'s interface. Thus a *requirement* is a function which associates a style-specific role with a number of operations, events or notifications which must be supported by the SMC who will be performing that specific role:

$$requirement : StyleRoles \rightarrow (Operations \cup Events \cup Notifications)$$

Roles in the local domain of an SMC combine both the *behaviours* and *requirements* of all architectural styles which were bound to them. For example, a given role can participate in an interaction with other SMCs simultaneously as an *inner* (through a *composition* structural style) and as a *source* (through a *diffusion* communication style). In this case, the role in question will thus accumulate the *behaviours* and *requirements* defined by the two styles.

An *architectural style* is thus defined as a set of style-specific roles, an associated behaviour and a set of requirements:

$$style = \langle StyleRoles, Behaviour, Requirements \rangle$$

where *style* is an architectural style, *StyleRoles* is the set of roles defined by this style, *Behaviour* is the implementation-specific behaviour defined by the style, 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 style.

Each architectural style can be additionally parameterised according to the abstraction it supports. This customisation involves either:

- what interface operations need to be filtered, mapped, etc (for a structural style); or
- what tasks and policies need to be loaded and in what conditions (for a management style); or
- what events need to be forwarded or subscribed to (for a communication style).

An *interaction specification* that is enforced by an SMC is defined by a number of architectural styles (and their respective style-specific roles), and how they are bound to roles defined within the SMC's local domain:

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

where *specification* corresponds to the interaction specification that an SMC must enforce, *Roles* is the set of roles defined in the local domain of this SMC, *Style* is the set of architectural styles bound to these roles through the set of bindings *Bindings*.

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

$$\begin{aligned} &binding(style, specification) \iff \\ &\qquad\qquad \forall\ x \in StyleRoles_{style}, \exists\ y \in Roles_{specification} :\ y := y\ \oplus\ x \end{aligned}$$

where the operator $\oplus$ applied to a style-specific role and a domain role adds to the latter the behaviour and requirements associated with the style-specific role. Associating an additional set of requirements with a domain role corresponds to 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 3.3.2.

When remote SMCs are discovered at run-time, they will be *assigned* to roles in the domain of another SMC if they satisfy the requirements for these roles, and the architectural styles which were previously bound will be instantiated. These will dictate how these SMCs should interact with each other (Figure 4.14).

Hence, an *assignment* of an SMC to a domain role within a specification will cause the deployment of all styles bound to this role:

$$assignment(SMC, role_{specification}) \iff$$
$$\forall\ st \in Styles_{specification} : \forall\ x \in StyleRoles_{st} : x \oplus^{-1} role_{specification} \rightarrow$$
$$deployment(SMC, st, x)$$

where the operator $\oplus^{-1}$ evaluates to *true* if an style-specific role and a domain role were previously *bound* to each other. This causes the deployment of the respective architectural style. The *deployment* of an architectural style is defined by the behaviour associated with the style, in terms of how the exchange of interfaces, policies or events is achieved. The deployment operation takes as arguments an SMC, the architectural style to be deployed and the specific role within the style that this SMC will be playing.

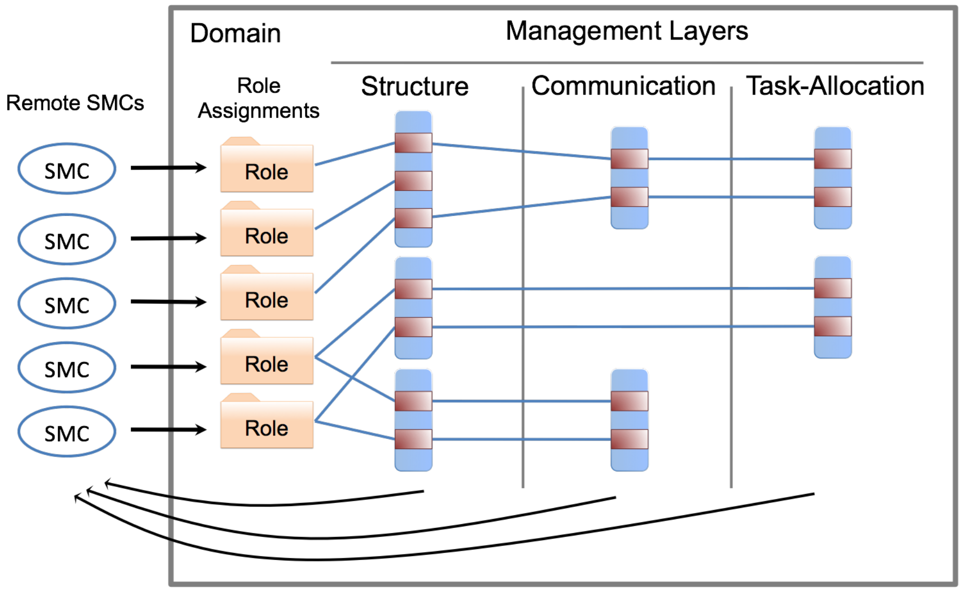


Figure 4.14: Composition model: (1) architectural styles are bound to *roles* in the local domain; (2) remote SMCs are assigned to these roles; (3) styles are deployed and the behaviour associated with each style is enforced in the SMCs

This composition model allows us to define layers of management for policy-based SMC interactions independently, where the structural, communication and management aspects can be specified by reusing common abstractions expressed as architectural styles. There are dependencies among the architectural styles: *structural styles* must be deployed first, as they enable the exchange of customised interfaces, e.g. *doctor* interface, *patient* interface; *communication styles* are then deployed to define patterns in terms of the events provided by these interfaces; *management styles* must be the last, as the policies loaded depend both on the operations provided by the application interface, as well as on the events forwarded by a communication style. The consistent use of architectural styles is discussed in Section 4.3.3.

4.3.2 Complex Styles and Distributed Enforcement

Certain combinations of architectural styles arranged in a particular manner will occur more often in specific application scenarios, e.g. for care management and physiological monitoring for a set of patients. For example, a *body-area network* is typically structured as a *composition* encapsulating the sensors and mediating (and filtering) access to its internal components. It normally relies on a *diffusion* event-forwarding scheme, where sensors forward events to the smartphone representing the patient SMC, which will possibly run other tasks that make use of the information monitored. In the following we present the notion of *complex styles*, and how different parts of an interaction can be instantiated by collaborating SMCs in a distributed manner.

Complex Styles

We define a *pattern of interaction* as a combination of architectural styles arranged for a particular purpose. In essence, a *pattern* is a complex style realised in terms of more primitive ones. A pre-defined pattern for a *body-area network*, for example, can then be instantiated multiple times to enforce the interactions among the sensors and devices available for each patient.

Figure 4.15 illustrates a succinct pseudo syntax for the textual description of patterns (in the current implementation patterns are written in *PonderTalk* which is more verbose – examples of patterns specified in the *PonderTalk* language can be found in Chapter 6). In *PonderTalk* each architectural style is also parameterised with the methods to be mapped or filtered, events to be forwarded or subscribed to, and missions to be loaded, however these details are not shown in Figure 4.15.

```
//Collaboration specification
type pattern ⟨$PatternName_0$⟩(...) {

  import /factory/structural/composition;
  import /factory/taskallocation/hierarchical;
  import /factory/communication/diffusion;

  type pattern ⟨$PatternName_1$⟩(role R1, [mandatory] role R2, role R3) {

   bind style composition(outer R1,inner R2,inner R3);
   [on ⟨event⟩] bind style hierarchical(manager R1,managed R2,
       managed R3);
   [on ⟨event⟩] bind style diffusion(target R1,source R2,source R3);
  }
  inst pattern $p_1$ = ⟨$PatternName_1$⟩($SMC_a$, $SMC_b$, $SMC_c$) at $SMC_1$;

           ...

   type pattern ⟨$PatternName_n$⟩(...) {
           ...
   }
   inst pattern $p_n$ = ⟨$PatternName_n$⟩(...) at $SMC_n$;
}

//Collaboration instantiation
inst pattern $p_0$ = ⟨$PatternName_0$⟩(...) at $SMC_0$;
```

Figure 4.15: Pseudo syntax for the textual representation of a pattern of interaction

The specification of a pattern is defined by a set of domain *roles* (amongst which some may be *mandatory* and others *optional*), a specific set of *architectural styles* expressing the abstractions required by this pattern, and *how the domain roles are bound to style-specific roles.* Bindings can be event triggered, meaning that they are only established in certain circumstances. When a pattern is instantiated, actual SMCs are passed as parameters and assigned to the respective roles. Mandatory roles must have been assigned when the pattern is instantiated, whereas optional roles can be discovered whilst the pattern is already running, e.g. in a *body-area pattern*, the *patient* and *heart rate sensor* roles are mandatory, whereas an *oxygen saturation* role may not be required for pattern instantiation and the respective SMC may be discovered later on.

Multiple patterns can be combined, possibly containing other nested patterns.

Each pattern is instantiated *"at"* an SMC, which is then responsible for instantiating the architectural styles defined within the pattern and establishing the respective interactions. This caters for the distributed enforcement of the whole interaction, where different SMCs are responsible for realising different parts of an interaction and each pattern can be independently specified and instantiated. This is discussed in more detail in the following.

Distributed Enforcement

In large applications, architectural styles and patterns will be established between different groups of SMCs. While a patient's smartphone will normally be responsible for enforcing a body-area pattern among its sensors, a doctor SMC will be for example interested in patterns for distributed monitoring of a set of patients. Similarly, a pattern running at the patient's home server will enforce an interaction between the local devices and appliances to define a home monitoring set-up SMC.

The interaction model based on architectural styles and patterns caters for the systematic specification and instantiation of SMC interactions in that: *(a)* it permits the establishment of different parts of an interaction (represented as sub-patterns) by different SMCs within a large collaboration; and *(b)* the internal specification of each pattern relies on a combination of abstractions for structure, management and communication that cater for the general organisation of SMCs inside that pattern.

Each pattern defines an interaction between a subset of SMCs and can use management and security strategies that differ from the ones used in another part of the interaction. For example, role assignment based on capabilities and authentication using public-key certificates (as illustrated in Section 3.6) can be used for managing parts of a larger interaction whereas the pattern enforced by another group of SMCs may rely on alternative role assignment and authentication strategies.

The SMC in charge of instantiating (part of) the *interaction specification* will be responsible for establishing the interactions with other participant SMCs. This consists of:

1. Determining who are the other participant SMCs. These can be:
   (a) SMCs which are discovered at run-time;
   (b) Pre-determined SMCs that parameterise the interaction specification;
2. Assign the SMCs to the roles defined in the interaction specification and verify whether these can be fulfilled by the SMCs, according to the role's *expected interface*;
3. Each participant will receive a subset of these roles, according to who it is *interacting* with (a role r_1 is *interacting* with a role r_2 if both r_1 and r_2 are bound to *style-specific* roles within the same architectural style), and the address of the SMCs assigned to these roles;
4. Each architectural style defined in the interaction specification is instantiated by the SMC in charge of enforcing the specification. This consists of:
   (a) Instantiating structural styles for interface exchange;
   (b) Instantiating communication styles for event exchange;
   (c) Instantiating management styles for policy exchange;
5. Repeat the whole process for any sub-pattern defined within the *interaction specification.*

Figure 4.16 illustrates the establishment of interactions based on patterns. In this example, SMC_0 is responsible for instantiating the *interaction specification* defined in $Pattern_0$. This pattern contains two individual architectural styles to be instantiated by SMC_0 between the participants SMC_1 and SMC_2, and also two other nested patterns. SMC_0 sends $Pattern_1$, which consists of a *composition* and a *hierarchical control* style, to SMC_1. Two nearby SMCs are discovered by SMC_1, and will be assigned to roles within the pattern if their interfaces fulfill the respective role's expected interfaces. SMC_1 will then instantiate the architectural styles within $Pattern_1$ between these SMCs. Similarly, SMC_0 sends

$Pattern_2$ to SMC_2. Each SMC which is responsible for enforcing a pattern is in charge of instantiating the styles within the pattern and establishing the respective interactions with other participating SMCs.

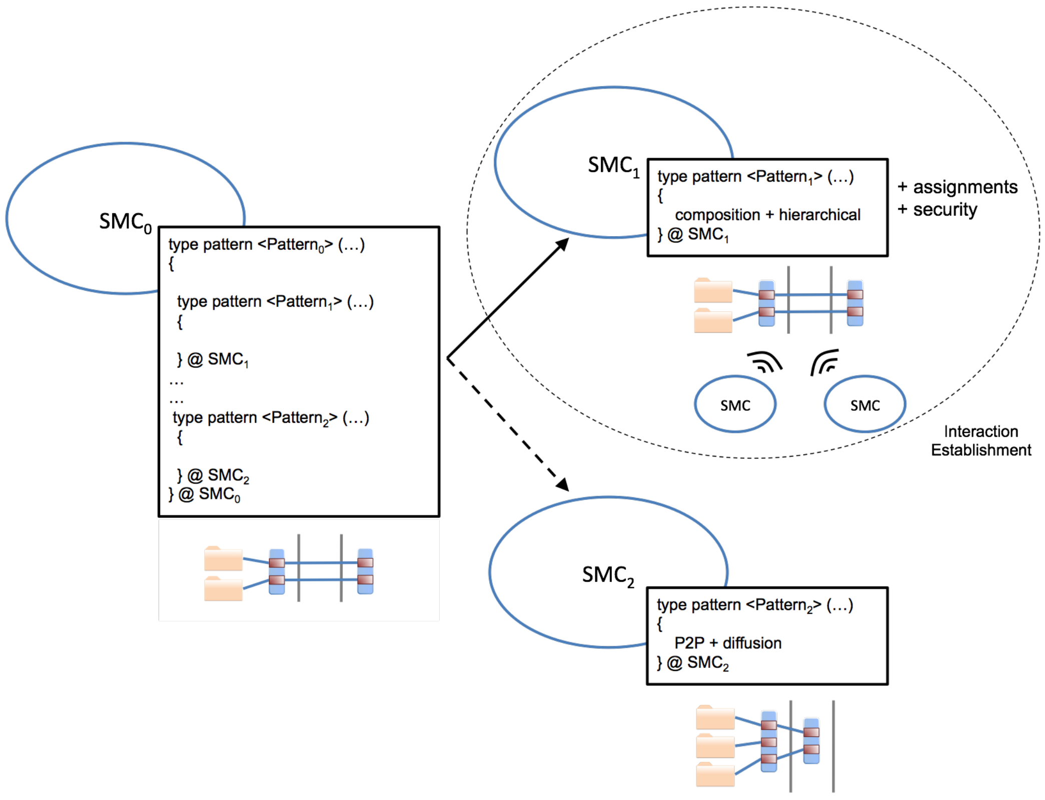

SMC0
type pattern <Pattern0> (...)
{
type pattern <Pattern1> (...)
{
} @ SMC1
...
...
type pattern <Pattern2> (...)
{
} @ SMC2
} @ SMC0
SMC1
type pattern <Pattern1> (...)
{
composition + hierarchical
} @ SMC1
+ assignments
+ security
SMC
SMC
Interaction Establishment
SMC2
type pattern <Pattern2> (...)
{
P2P + diffusion
} @ SMC2

Figure 4.16: Distributed enforcement of architectural styles and patterns

When an architectural style is instantiated by an SMC, this will distribute different parts of the algorithm or protocol implemented within the style to the other participant SMCs, according to their roles. By executing their fragments of the algorithm or protocol, the collection of interacting SMCs will collaboratively enforce the semantics of the respective architectural style. In Chapter 6 we will give details on the runtime model that supports the instantiation of patterns and the establishment of interactions between distributed SMCs, and how these were implemented within the framework.

The use of styles and patterns provides a much richer model for the specification, instantiation and reuse of SMC interactions. However, because different parts of a large interaction will be enforced by different SMCs, these collaborations are more susceptible to inconsistencies that will prevent SMCs from operating as originally expected. The correct specification and deployment of SMC interactions are discussed in the next section.

4.3.3 Correct Specification and Deployment

Interactions between SMCs can be subject to a number of inconsistencies, both during the interaction specification and during the interaction instantiation. These inconsistencies are typically related to *bindings*, *assignments* and *policies* within the context of an interaction.

Bindings

The way an architectural style is bound to a set of application roles defines how the SMCs assigned to these roles will be expected to interact. If these bindings do not respect the semantics of each style, the resulting specification may be flawed. For example, with respect to the structure of an interaction, an SMC encapsulated in a *composition* style should not be visible or accessible by SMCs outside the composition.

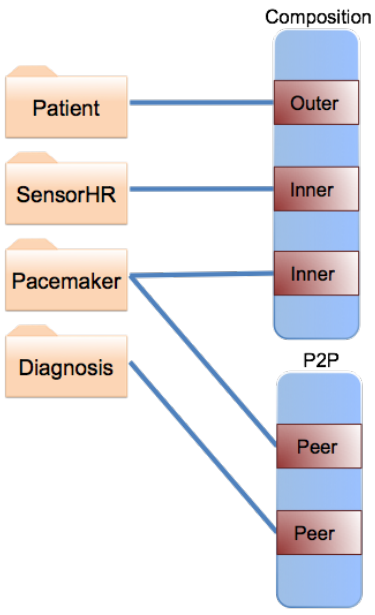


Figure 4.17: Incorrect binding of roles

Figure 4.17 illustrates an example of an incorrect specification in terms of bindings. In this case, the *Patient* role is expected to establish a composition with the *Pacemaker* role, which is thus bound to an *inner* style-specific role. However, the same *Pacemaker* role is also bound to a *peer* role in a *peer-to-peer* interaction with an external diagnosis device. This violates the semantics of encapsulation (where an *inner* SMC can only be accessed by its *outer* SMC, and all invocations or event notifications come via the outer's interface) defined by the composition architectural style.

Inconsistent bindings also happen across styles from different categories: for example, an event-forwarding style can only be bound if a style that first exchanges the interfaces required for supporting these events is already bound.

Assignments

Even if an interaction is correctly specified in terms of bindings, the assignment of SMCs that do not provide suitable interfaces will impede the deployment of the interaction. Thus the SMCs assigned to each role must satisfy the *requirements* defined by all the styles bound to that role in terms of operations, events or notifications.

In addition, SMC assignments may also violate style semantics, even if the bindings were correctly specified. Figure 4.18 illustrates an example of inconsistent SMC assignment, where both a *composition* and a *peer-to-peer* architectural styles are bound to a set of domain roles. Although the bindings between roles do not violate the semantics of encapsulation defined by the composition style, SMC_3 is assigned both to the *Pacemaker* role (which is bound to the *inner* style-specific role of the composition) and to the *Actuator* role (which is bound to a *peer* style-specific role of the peer-to-peer style). Even though the bindings were specified correctly, this set of assignments caused the same SMC to participate in a composition (as an *inner*) and in a peer-to-peer interaction with another SMC, thus violating the semantics of encapsulation.

The verification of SMC assignments can be seen as *run-time checks* that must

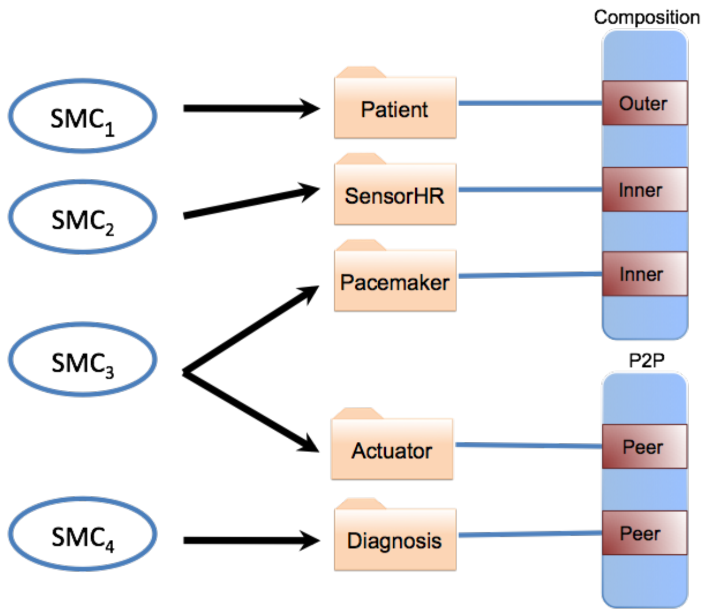


Figure 4.18: Incorrect assignment of SMCs

be performed when actual SMCs are discovered, whereas the verification of the bindings can be seen as *static checks* of the interaction specification.

Policies

Finally, after all architectural styles are bound and deployed across a set of SMCs, a policy-based collaboration must be effectively achieved. However, if patterns are specified for loading policies into a specific SMC but this SMC does not have access to the events required for triggering the policies (or these events are never forwarded to the SMC), then the policies will never be triggered. Similarly, an SMC may not have access to the interfaces required for validating the actions prescribed by a policy.

Thus, additional checks must guarantee that the exchanged policies were accompanied by the adequate exchanges of events and interfaces required by these policies. In the next chapter, we will describe how model-checking techniques were applied to analyse and verify the consistency of policy-based SMC interactions automatically.

4.4 Discussion

Engineering large-scale policy-based systems using simple abstractions such as policies, roles, interfaces and events can be difficult to manage and deploy. The use of architectural styles and patterns to specify, instantiate and reuse SMC interactions is a promising approach which incorporated ideas from software architecture-based approaches [GS93, SDK$^+$95, SDZ96, MMP00, MT00, TMD09]. We proposed a catalogue of architectural styles that promotes the reuse of common abstractions for the structure, management and communication aspects for SMC interactions. Collaborations of SMCs can interact with other collaborations, and re-apply architectural styles recursively. This facilitates defining large-scale composable systems by reusing and combining common abstractions. Chapter 6 will present details on how these architectural styles are implemented and Chapter 7 will describe a case-study illustrating the use of SMCs and architectural styles in the design of a healthcare monitoring application.

The different categories of architectural styles can be seen as complementary perspectives for modeling policy-based SMC interactions. The proposal of a catalogue of styles brings to light common types of interaction that are useful for building collaborations between SMCs. However, this chapter does not present an exhaustive catalogue and only focuses on the frequently occurring patterns identified in a few application scenarios. Investigation of additional scenarios and applications may uncover new patterns.

Our overall goal is to be able to dynamically form collaborations, compositions and federations of SMCs suitable to particular application scenarios, e.g. care management for a set of patients, by instantiating combinations of pre-defined patterns. These application-specific patterns rely on the primitive architectural styles and can be parameterised and instantiated for each patient SMC, using the resources and devices available in a body-area network or in a home monitoring set-up, for example.

Although architectural styles proved a helpful abstraction for designing large-scale SMC interactions, the successful operation of these collaborations depends both on their correct specification and on the suitability of the participating SMCs. The next chapter will present a formal specification of SMC behaviour and interactions, and how model-checking techniques can be applied for verifying correct interaction specification and deployment.

Chapter 5

Formal Specification and Model-Checking

This chapter presents a formal specification of the overall SMC behaviour and its analysis in collaborations across SMCs. In these collaborations, consistent policy deployment is crucial as often SMCs form autonomous administrative domains and, when these SMCs are composed or federated, inconsistencies, conflicting policies or unsuitability of the resources available will 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.

We chose the *Alloy Analyzer* [Jac02, Jac06] 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 [MPW92], ambient calculus [CG98] and channel ambient calculus [Phi06] which model the computation operationally. We found *Alloy* more natural and concise for describing SMC interactions and the integrity constraints related to SMC management.

Models written in *Alloy* can be automatically checked for correctness using its analyser. *Alloy* 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 reducible to a boolean formula). This is based on the *small scope hypothesis* [Jac06], that for any flawed design a counter-example should be found by an exhaustive search within a comparatively small, bounded scope.

The formal model presented in this chapter complements our framework for the specification and establishment of SMC interactions. The tool support provided by the *Alloy Analyzer* allows:

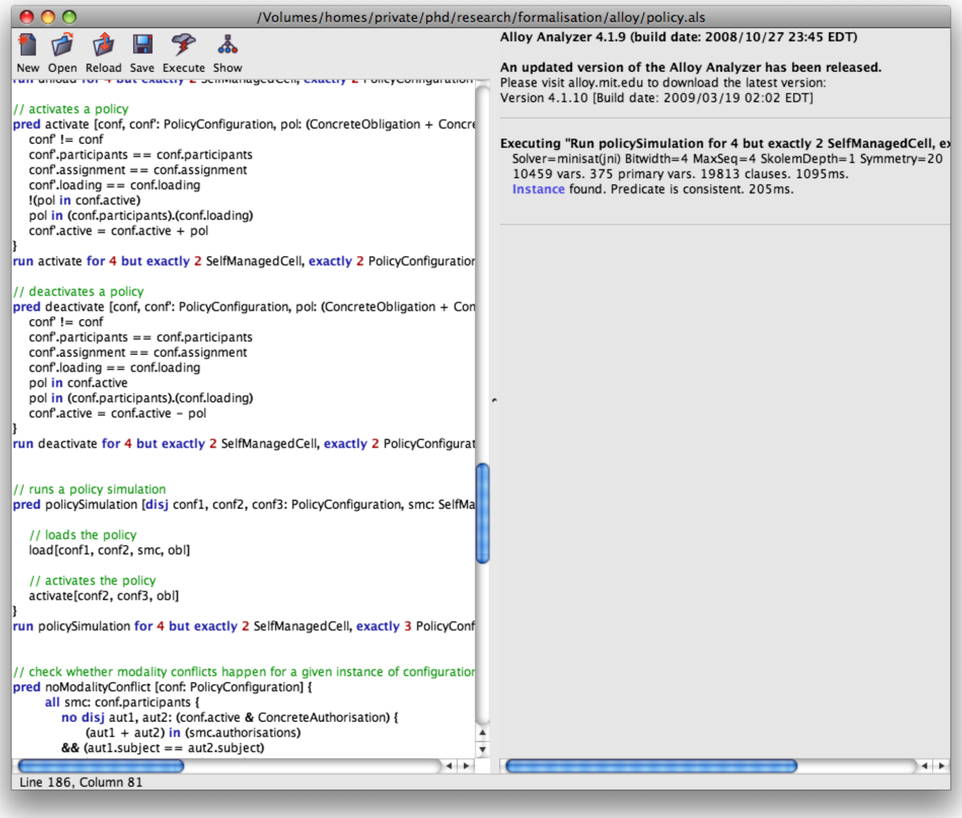
(a) formally capturing the static and dynamic aspects of the structure (through *signatures*) and behaviour (through *predicates*) of a model for SMC interactions;

(b) automatic verification of the consistency of specific collaborations between SMCs by using its analyser;

(c) simulation of SMC behaviour in complex interactions involving pre-determined sequences of operations.

The model specification formally defines SMCs and interrelated concepts, e.g. roles, interfaces, etc, and the SMC's behaviour with respect to establishing policy-based interactions with other SMCs, e.g. discovery, assignment, etc. The model also logically defines properties that can be verified for SMC interactions, e.g. an SMC satisfies the requirements for a role. We can then check whether a specific set of SMCs and their policies (given as input) satisfy these logical properties. The boolean evaluation of these properties is not affected by the size of the interactions, rather it depends on whether these interactions conform to a number of logical statements.

The formal model and the tool support are used to design and to check whether SMC interactions can be established before implementing them and deploying the policies in actual devices. A set of SMCs and their interfaces is given as input to the model, which checks whether they are capable of establishing the


/Volumes/homes/private/phd/research/formalisation/alloy/policy.als
New
Open
Reload
Save
Execute
Show

```
// activates a policy
pred activate [conf, conf': PolicyConfiguration, pol: (ConcreteObligation + Concre
   conf' != conf
   conf'.participants == conf.participants
   conf'.assignment == conf.assignment
   conf'.loading == conf.loading
   !(pol in conf.active)
   pol in (conf.participants).(conf.loading)
   conf'.active = conf.active + pol
}
run activate for 4 but exactly 2 SelfManagedCell, exactly 2 PolicyConfiguratior

// deactivates a policy
pred deactivate [conf, conf': PolicyConfiguration, pol: (ConcreteObligation + Con
   conf' != conf
   conf'.participants == conf.participants
   conf'.assignment == conf.assignment
   conf'.loading == conf.loading
   pol in conf.active
   pol in (conf.participants).(conf.loading)
   conf'.active = conf.active - pol
}
run deactivate for 4 but exactly 2 SelfManagedCell, exactly 2 PolicyConfigurat


// runs a policy simulation
pred policySimulation [disj conf1, conf2, conf3: PolicyConfiguration, smc: SelfMa

   // loads the policy
   load[conf1, conf2, smc, obl]

   // activates the policy
   activate[conf2, conf3, obl]
}
run policySimulation for 4 but exactly 2 SelfManagedCell, exactly 3 PolicyConf


// check whether modality conflicts happen for a given instance of configuratior
pred noModalityConflict [conf: PolicyConfiguration] {
      all smc: conf.participants {
          no disj aut1, aut2: (conf.active & ConcreteAuthorisation) {
                (aut1 + aut2) in (smc.authorisations)
          && (aut1.subject == aut2.subject)
```

Line 186, Column 81

```
Alloy Analyzer 4.1.9 (build date: 2008/10/27 23:45 EDT)

An updated version of the Alloy Analyzer has been released.
Please visit alloy.mit.edu to download the latest version:
Version 4.1.10 [Build date: 2009/03/19 02:02 EDT]

Executing "Run policySimulation for 4 but exactly 2 SelfManagedCell, ex
   Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   10459 vars. 375 primary vars. 19813 clauses. 1095ms.
   Instance found. Predicate is consistent. 205ms.
```

Figure 5.1: Alloy analysis tool screenshot

interactions and enforcing their policies. These *a priori* checks cannot guarantee the correct enforcement of the interaction during run-time in which, for example, a sensor may fail and the interaction may have to be re-checked. Even though it would be possible to use *Alloy* to re-check such an interaction dynamically, currently we only use the tool for performing design-time checks. The tool can also be used to simulate SMC behaviour in complex interactions. For example, given a particular configuration of doctor and patient SMCs, we can simulate the discovery of a new sensor by the patient, then the loading of a policy into the patient and the subsequent failure of another sensor. We can add further steps in specific *guided simulations* [HR04] and verify properties in different stages of this interaction.

Figure 5.1 illustrates a screenshot of the *Alloy Analyzer*: on the left hand side the editor allows the specification of the model itself and the logical properties we want to verify. Predicates can be defined that specify a particular instance of SMC interaction and whether this interaction satisfies a number of logical properties. The results of the analyses are presented in the panel on the right hand side. The toolset also provides a visualisation tool which can be used to display examples or counter-examples graphically. This has helped us in understanding the solutions found by the analyser. The figures in this chapter were generated by this visualiser, with small hand edits of names to aid comprehension.

The next sections will first describe the formal specification of SMCs and their interactions, and then how analyses and verification of properties can be performed using this formal model.

5.1 Modelling Structure and Behaviour

This section presents the specification of the formal model for SMCs. We concentrate on the specification of the *structure* and *behaviour* of the model. The structure is determined by a set of *signatures*, which define the concepts relevant to the model and their relation to other concepts, e.g. SMC, role, interface. Signatures refer to the structure of the model specification, which is similar to a class structure in the OO paradigm. This should not be mistaken for the structural aspects of an interaction as defined by the structural architectural styles in Chapter 4. The dynamic behaviour of the model is determined by a set of *predicates*, which specify the effect of operations executed in the model, e.g. SMC discovery, role assignment. The formal specification also consists of a number of sanity constraints specified in *Alloy* in the form of *facts*. Facts define properties that always hold in the model, for example, an interface is always provided by one SMC so interfaces do not just occur unattached to any SMC. These facts however are almost always trivial constraints and therefore are omitted from this section, which instead focuses on the specification of the structure and behaviour of the model alone.

In order to facilitate its understanding, the specification of the formal model is divided into two parts:

- Firstly, the ***basic SMC model*** defines the elementary concepts such as SMC, Role, Interface, policies and the basic predicates for discovery of a new SMC, departure of an SMC, role assignment and role de-assignment, as well as the use of obligation and authorisation policies in collaborations of SMCs;
- Secondly, the ***architectural model*** formalises the architectural aspects of an interaction, such as architectural styles, bindings, etc, and shows how architectural styles can be composed to form larger interactions.

5.1.1 Basic SMC Model

This model defines the elementary principles for policy-based interactions between SMCs, without yet considering the use of architectural styles for composing these interactions. This forms the basis of a complete specification for modelling SMC behaviour and interactions.

Structure

The most important component in the model is the SMC, which is modelled in the signature *SelfManagedCell* (Figure 5.2). In the declaration of a signature body, a number of relations are defined, which can be thought of as fields of an object in the OO paradigm. The *SelfManagedCell* signature specifies four *unary relations*[1]. The first two, *provides* and *requires*, define respectively which *interfaces* an SMC is able to offer to remote SMCs and which *roles* an SMC requires to be fulfilled. The other two relations, *obligations* and *authorisations*, define the policies that an SMC is enforcing. *SelfManagedCell* is an **abstract** signature, meaning it can be extended to define a specialised component in the

[1]In *Alloy*, a relation is defined as a set of ordered tuples, and the arity of the relation is the number of elements in each tuple. Each tuple thus indicates that its elements are related in a certain way, e.g. the arity-2 relation *"assignment: Interface → Role"*, which will be discussed later, associates interfaces with roles.

model, e.g. *DoctorSMC*, *PatientSMC*, *SensorSMC* all extend the abstract signature *SelfManagedCell*.

```
abstract sig SelfManagedCell
{
    provides: some Interface,
    requires: some Role,
    obligations: set Obligation,
    authorisations: set Authorisation
}
```

Figure 5.2: Self-Managed Cell signature

An SMC provides one or more interfaces, which can then be assigned to a role in a remote SMC. The signature *Interface* (Figure 5.3) 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 an interface. In turn, these are defined in the *Operation*, *Event* and *Notification* signatures (not shown here).

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

Figure 5.3: Interface signature

A *Role* (Figure 5.4) functions as a placeholder for remote SMCs. The relation *"assignment: Interface → Role"* (which will be discussed shortly) is used to define the assignment of an SMC's interface to a role. The role's *expected interface* (requirements that must be fulfilled by interfaces assigned to the role) is determined by the set of architectural styles which are bound to that role (this will be discussed in Section 5.1.2).

```
abstract sig Role
{ }
```

Figure 5.4: Role signature

An SMC enforces two types of policies: the *ConcreteObligation* signature (Figure 5.5) defines the subject and target roles that the policy refers to, 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
}
```

Figure 5.5: Concrete obligation signature

Similarly, the *ConcreteAuthorisation* signature (Figure 5.6) defines a subject role, a target role, an action and the modality of the policy (which can be either positive or negative). In both types of policies, the subjects and the targets are roles within the context of the SMC in which the policy is enforced.

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

Figure 5.6: Concrete authorisation signature

Behaviour

Dynamic behaviour is modelled in Alloy through a number of *predicates*. A common technique [HR04, Tut09] to represent dynamic behaviour is to define declaratively how the occurrence of an operation affects the state of the system being modelled. It is necessary to represent explicitly the state of the system being modelled in Alloy, so we can show what properties hold before and what properties hold after an operation is executed.

Typically an additional signature is used to represent the state of the system, which in our case corresponds to an interaction between a set of SMCs. Then, for each operation we define a predicate that takes as arguments an instance *S* and an instance *S'* of this signature (to represent the *"state"* of the interaction before and after the execution of the operation), and shows how *S* differs from *S'* in this predicate. This is equivalent to showing which properties hold before and which properties hold after the execution of the operation. This notion of *"state"* of an interaction was encoded in the signature *Configuration* (Figure 5.7).

```
sig Configuration
{
    participants: some SelfManagedCell,
    assignment: Interface lone → Role,
    loading:  SelfManagedCell →
                (ConcreteObligation + ConcreteAuthorisation),
    active: set (ConcreteObligation + ConcreteAuthorisation)
} {
    active in (participants.obligations
             + participants.authorisations
             + participants.loading)
}
```

Figure 5.7: Configuration signature

An instance of this signature thus represents a policy interaction between a set of SMCs at a given time point. The signature specifies four relations: *(a)* the set of *participants* in the interaction, *(b)* the *assignments* of participants (represented by their provided interfaces) to roles, *(c)* the policies that are exchanged (*loading*) between these SMCs, and *(d)* the policies that are currently *active* in each of the participating SMCs. An appended fact in this signature ensures that the active policies within a configuration are either the policies already inside one of the participants or the policies being loaded.

We specified the various operations required for modelling SMC behaviour as *predicates*, that show which properties hold before and which properties hold after the execution of the operation. Using this principle it was possible to show the changes that happen when an SMC is discovered, when an SMC departs, when an SMC is assigned to a role, when architectural styles are bound to roles and when these styles are deployed. The *basic SMC model* caters for the specification of:

- Discovery of an SMC;
- Departure of an SMC;
- Assignment of an SMC to a role;
- De-assignment of an SMC from a role;
- Loading and activation of a policy;
- Unloading and de-activation of a policy.

Figure 5.8 illustrates the operation of *assignment* of interface *itf* of an SMC *smc1* to a role *rol* of another SMC *smc2*. It defines the *assign* operation where *conf* and *conf'* denote the interaction before and after the operation is executed, respectively. It states that both *smc1* and *smc2* must already be participants in *conf* (line 5), and that *smc1* must provide *itf* whereas *smc2* must require *rol* in *conf* (lines 6-7). The operation causes the set of assignments in the new configuration *conf'* to be the same as the one before the operation plus the new assignment *itf* → *rol* (line 11). Section 5.1.2 will discuss how the *requirements* associated with a role are checked against the interface assigned to it.

```
1  pred assign [disj conf, conf’: Configuration,
2                     smc1: SelfManagedCell, itf: Interface,
3                     smc2: SelfManagedCell, rol: Role]
4  {
5       (smc1 + smc2) in conf.participants
6       itf in smc1.provides
7       rol in smc2.requires
8       conf.participants  = conf’.participants
9       conf’.loading = conf.loading
10      conf’.active = conf.active
11      conf’.assignment = conf.assignment + (itf → rol)
12 }
```

Figure 5.8: Assign predicate

The assign operation does not affect the *participants*, *loading* and *active* properties of the interaction (lines 8-10); rather, these properties are modified by separate predicates, which were specified in a similar manner. These individual predicates can then be combined to *simulate SMC behaviour*, for example, to represent the assignment of an SMC to a role followed by policy loading. This can be done by defining a new predicate which combines *assign* to show how the interaction initially passes from *conf* to *conf'*, and *load* to show how this interaction then passes from *conf'* to *conf''*. The final configuration *conf''* will thus represent the resulting *"state"* of the interaction after the two operations are executed.

Figure 5.9 shows an example of a valid SMC interaction which can be obtained from the Alloy specification. In this example, *PatientSMC* requires the roles *Sensor* and *Doctor*, and provides interface *IPatient* (which defines the operation *startECG*). Similarly, *DoctorSMC* requires the role *Patient*, and provides interface *IDoctor* (which defines the operation *load*, the event *loaded* and the notification *stopped*). In the example, interface *IPatient*, which is provided by *PatientSMC*, is assigned to the role *Patient*, which is required by *DoctorSMC*.

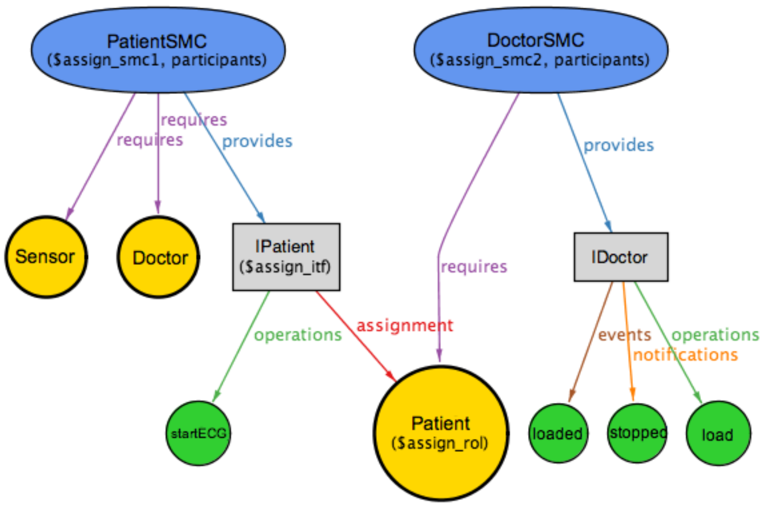


Figure 5.9: Alloy graphical representation of a role assignment

Another example, this time describing a policy interaction across SMCs is shown in Figure 5.10. *DoctorSMC* has an obligation policy *Obl*, which states that the subject role (*Doctor*) must invoke on the target role (*Patient*) the action *startECG* in response to the event *highHR*. Interface *IDoctor* provided by the *DoctorSMC* is locally assigned to the subject role, and interface *IPatient* provided by the remote *PatientSMC* is assigned to the target role in the *DoctorSMC*.

PatientSMC enforces the positive authorisation *Aut* (labelled *"modality: Positive"*), which states that the subject role (*Doctor*) is allowed to invoke the action *startECG* on the target role (*Patient*). The remote interface *IDoctor* provided by the *DoctorSMC* is this time assigned to the *Doctor* role in *PatientSMC*. In this example the same interface the doctor uses is exported to be seen by the patient. In *PatientSMC*, the local interface *IPatient* is assigned to the local *Patient* role, which is also the target role of the policy being enforced by this SMC. Both *Obl* and *Aut* policies are active in this interaction.

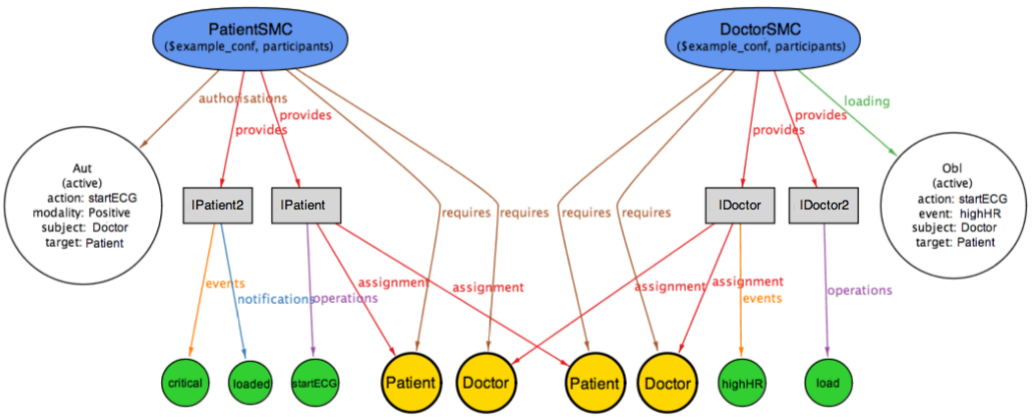


Figure 5.10: Alloy graphical representation of a policy deployment between SMCs

Examples of the types of analysis we are able to perform using this formal specification will be discussed in Section 5.2. Before that, we describe the formalisation of the architectural aspects of an interaction.

5.1.2 Architectural Model

We defined the architectural aspects of SMC interactions, in particular styles and bindings between style-specific and domain roles, in a separate model. This distinction between the fundamental concepts and the architectural aspects of SMC interactions permits independent modelling and analysis of either simple interactions or more complex interactions based on the use of styles. The *architectural model* extends the *basic SMC model* with the signatures and predicates necessary for the specification of these additional concepts. It supports reasoning about SMC interactions based on the composition of architectural styles, thereby allowing the verification of properties related to consistent bindings and deployment of styles.

Structure

Architectural styles were modelled in Alloy using a three-level hierarchy, which distinguishes between *(a)* the concerns that are common to all architectural styles, *(b)* the concerns that are related to a specific category of architectural styles, and *(c)* the concerns that are specific to a particular abstraction enforced by an architectural style:

- The top-level signature *ArchitecturalStyle* is common to all styles, independent of their category or the specific abstraction they support.
- Three other intermediate signatures, *StructuralStyle*, *CommunicationStyle* and *TaskAllocationStyle*, are used to represent the mapping of interfaces, forwarding of events and loading of policies respectively.
- Each style-specific signature then defines further constraints on how this is achieved, i.e. how mappings should be performed, how events should be forwarded and how policies should be loaded.

The signature *ArchitecturalStyle* (Figure 5.11) is the base signature of any architectural style. It defines two relations: *roles* which specifies a set of style-specific roles (defined by the signature *ArchitecturalStyleRole*, not shown here), and *expects* which associates a style-specific role with a set of requirements (in terms of operations, events or notifications).

```
abstract sig ArchitecturalStyle
{
    roles: some ArchitecturalStyleRole,
    expects: roles → (Operation + Event + Notification)
}
```

Figure 5.11: Base architectural style signature

The exchanges of interfaces, events and policies were abstracted in three additional signatures that extend *ArchitecturalStyle*, namely *StructuralStyle*, *CommunicationStyle* and *TaskAllocationStyle*, referred to as intermediate signatures. Figure 5.12 shows the three signatures, and how we *modelled*[2] the exchanges of interfaces, events and policies in terms of *(1)* the *mapping* of operations of an interface, *(2)* the *forwarding* of events generated by an SMC to notifications received by another SMC, and *(3)* the *loading* of policies[3] into an SMC performing a given style-specific role.

```
abstract sig StructuralStyle extends ArchitecturalStyle
{
    mapping: Operation → Operation
}

abstract sig CommunicationStyle extends ArchitecturalStyle
{
    forwarding: Event → Notification
}

abstract sig TaskAllocationStyle extends ArchitecturalStyle
{
    loading: ArchitecturalStyleRole → Obligation
}
```

Figure 5.12: Intermediate structural, communication and task-allocation style signatures

Finally, each individual architectural style extends one of the three intermediate signatures to define specific constraints on how this behaviour is achieved. The definition of each architectural style is accompanied by the specification of additional signatures that represent the style-specific roles (not shown here) which are pertinent for a given style, e.g. *Inner* and *Outer* are empty signatures that extend *ArchitecturalStyleRole*, and are used specifically in the context of a composition. Figure 5.13 illustrates how the signature *Composition*, which extends *StructuralStyle*, was defined. Rather than explaining in detail the syntax we present in Table 5.1 an intuitive description of the most common operators

[2]The Alloy specification abstracts some of the details when modelling the architectural styles; for example, a structural styles is more generally modelled as a *mapping* of operations, which corresponds to a mapping from the *Outer* to the *Inner* (in *Composition*), or an empty mapping (in *Peer-to-Peer*) for example. Other forms of behaviour, such as the *filtering* of operations were deliberately omitted to make the model simpler to understand.

[3]In our model, the set of policies loaded by a task-allocation style corresponds to the *mission* specification which is sent to an SMC.

```
1  sig Composition extends StructuralStyle
2  { }
3  {
4      roles in (Outer + Inner)
5      #(roles & (Outer)) == 1
6      #(roles & (Inner)) >= 1
7      all conf: ArchitecturalConfiguration |
8          (this in conf.bound)
9              ⇒ no (((roles & Outer).(conf.binding)
10                 & (roles & Inner).(conf.binding)))

12     all conf: ArchitecturalConfiguration |
13         (this in conf.deployed)
14             ⇒ no (((roles & Outer).(conf.binding).~(conf.assignment)
15                 & (roles & Inner).(conf.binding).~(conf.assignment)))

17     // requirements
18     expects in ((roles & (Inner)) → Operation)

20     // behaviour
21     all conf: ArchitecturalConfiguration | (this in conf.deployed)
22         ⇒ mapping in
23     (((roles & (Outer)).(conf.binding)).~(conf.assignment).operations)
24  → (((roles & (Inner)).(conf.binding)).~(conf.assignment).operations)
25 }
```

Figure 5.13: Composition signature

used in this example for those not familiar with *Alloy*. The style specification does not define new relations in its body, but instead simply adds further constraints through a number of appended facts, just after the signature body. The definition of this style relies on an *ArchitecturalConfiguration*, which represents an instance of an interaction at a given time point, similar to a *Configuration* (Figure 5.7). In particular, *"binding"* is a relation *"ArchitecturalStyleRole → Role"* (defined in Figure 5.15) that associates a given style-specific role with one of the roles required by an SMC. The relations *"bound"* and *"deployed"* define which architectural styles were bound in this interaction, and among those, which ones are already instantiated/deployed in the participant SMCs. Firstly, the specification restricts the style-specific roles for a composition: style-specific roles can be either *Outer* or *Inner*, with exactly one instance of the former and one or more instances of the latter (lines 4-6). Then two further constraints are added which state that the bindings (resp. assignments) of the *Outer* role must be disjoint from the bindings (resp. assignments) of the *Inner* roles, i.e. an

Table 5.1: Common Alloy operators

Operator	*Description*
Intersection ("&")	Defines the intersection between two sets, thus *"(roles & (Inner))"* in line 6 of Figure 5.13 returns the set of all style-specific roles which belong to that style and which are of the type *Inner*.
Relational inverse ("∼")	Used to reverse a relation, so given for example the relation *"conf.assignment: Interface → Role"*, which can be used to obtain the *Role* to which a given *Interface* was assigned, *"∼conf.assignment"* inverts that relation to *"Role → Interface"*, which can be used to obtain the *Interface* assigned to a given *Role*.
Relational join (".")	Used to compose two relations. For example, the expression defined in line 23 of Figure 5.13 corresponds to the set of all *"operations"* provided by the interfaces which are assigned to the set of roles ("*∼(conf.assignment).operations*"), which are bound to the *Outer* style-specific role (*"((roles & (Outer)).(conf.binding)"*) in this composition. In this case, *"binding"* is a relation *"Architectural-StyleRole → Role"* (defined in Figure 5.15) which can be used to obtain the domain role to which a given style-specific role was bound.

SMC cannot be outer and inner at the same time in a composition (lines 7-15). The composition signature also constraints the requirements of the style to a set of operations that are associated with the *Inner* style-specific role (line 18). Finally, the composition defines the behaviour that must be added when the style is deployed, in the form of mapping of operations from the interface of the *Outer* SMC to the interface of the *Inner* SMC (lines 21-24).

The enforcement of encapsulation in a *Composition* is carried out during the binding operation (as part of the model behaviour, which will be discussed shortly). This checks that the addition of the new bindings to the current interaction will not result in a set of inconsistent bindings (when the system changes from a *"state" S* to a *"state" S'*). The condition that verifies whether the encapsulation property is satisfied is illustrated in Figure 5.14. It specifies that a new structural style *"st"* can only be bound if the following applies for any other structural style *"otherstyle"* which is already bound. Let *conf* and *conf'* be the representation of the interaction before and after the new binding happens respectively:

1. If *"st"* is a composition (line 3), then there is no intersection between the resulting bindings of its inner role (line 4) and the existing bindings of any *"otherstyle"* (line 5), except if this overlapping happens with an *Outer* role of *"otherstyle"*. In this latter case the intersection would result in a multi-level composition, which preserves encapsulation and is therefore a valid structure; and
2. If *"otherstyle"* is a composition (line 8), then there is no intersection between the existing bindings of its inner role (line 9) and the bindings of the new style (line 10), i.e. new bindings are not overlapping with roles that were *already* encapsulated.

```
1  all otherstyle: (conf.bound & StructuralStyle)
2  {
3      (!(no (st & Composition))
4        ⇒ no ((st.roles & Inner).(conf'.binding)
5          & ((otherstyle.roles).(conf'.binding) −
6            ((otherstyle & Composition).roles & Outer).(conf'.binding))))
7   and
8      (!(no (otherstyle & Composition))
9        ⇒ no ((otherstyle.roles & Inner).(conf'.binding)
10         & (st.roles).(conf'.binding)))
11 }
```

Figure 5.14: Verifying if encapsulation is preserved during binding

Other architectural styles were similarly modelled in Alloy, thus allowing us to define specific abstractions for interface mappings, event forwarding and policy loading.

Behaviour

The behavioural aspects of the architectural model concentrate on two distinct steps related to SMC interactions: *(1) interaction specification* and *(2) interaction instantiation. Interaction specification* is defined by the bindings between style-specific roles and the roles required by an SMC, and this specifies how SMCs assigned to these roles will be expected to behave. *Interaction instantiation* is represented by the deployment of styles previously bound to a set of roles, when actual SMCs are assigned to these roles. The deployment causes a new dynamic behaviour to be added to the current interaction (when the system passes from state *S* to *S'*), which is defined in terms of forwarding of events, mapping of interfaces or exchange of policies, depending on the architectural style being deployed.

The predicates defined in this model cater for the specification of:

- Binding and unbinding of an architectural style;
- Deployment and removal of an architectural style.

To distinguish the state of the system before the execution of an operation from the state after the operation, we defined an additional signature, named *ArchitecturalConfiguration* (Figure 5.15). Similarly to the predicates defined in the *basic SMC model*, the various operations for binding and deployment of architectural styles were modelled by showing how instances of *Architectural-Configuration* differ before and after an operation is executed.

```
sig ArchitecturalConfiguration extends Configuration
{
    patterns: some Pattern,
    bound: set ArchitecturalStyle,
    deployed: set ArchitecturalStyle,
    binding: ArchitecturalStyleRole → lone Role,
    forwarding: Event → Notification,
    loading: SelfManagedCell → Obligation,
    mapping: Operation → Operation
} {
    bound in patterns.styles
    deployed in bound
    binding in (bound.roles → lone participants.requires)
}
```

Figure 5.15: Architectural configuration signature

An *ArchitecturalConfiguration* extends *Configuration* (defined in Figure 5.7) and includes the architectural aspects of an interaction through a number of additional relations:

1. a set of *patterns* involved in this interaction (the *Pattern* signature consists of a set of architectural styles, and was omitted from this discussion);
2. a set of architectural styles currently *bound*;
3. a set of *bindings* which bind style-specific roles to SMC roles;
4. a set of architectural styles currently *deployed*; and
5. a behaviour added by each style, in the form of:
   (a) *forwarding* of events;
   (b) *loading* of policies; or
   (c) *mapping* of operations provided by an interface.

The appended facts (lines 11−13) constrain these relations by specifying that bound styles can only be the ones defined within the patterns involved in this interaction, that deployed styles can only be among the ones which were already bound, and that the bindings must be specified in terms of the roles belonging to the bound styles and the roles required by the participants of this interactions only.

Figure 5.16 shows an architectural configuration obtained from the Alloy model. In this example interaction *PatientSMC* requires two roles, *Patient* and *Sensor*, and these application roles are bound to a number of style-specific roles. The *Sensor* role is bound to *Source* (through a *Diffusion* style), *Managed* (through a *HierarchicalControl* style) and *Inner* (through a *Composition* style). Similarly, the *Patient* role is bound to *Target* (through a *Diffusion* style), *Manager* (through a *HierarchicalControl* style) and *Outer* (through a *Composition* style). This architectural configuration means that whichever SMCs are assigned to the *Patient* and *Sensor* roles required by *PatientSMC*, they will behave according to the respective parts defined by each architectural style bound to their roles. Each architectural style specifies *(a)* what it *expects* from the SMCs that will be eventually assigned to the roles, as well as *(b)* the *behaviour* to be added to the interaction after the style is deployed (forwarding, loading or mapping).

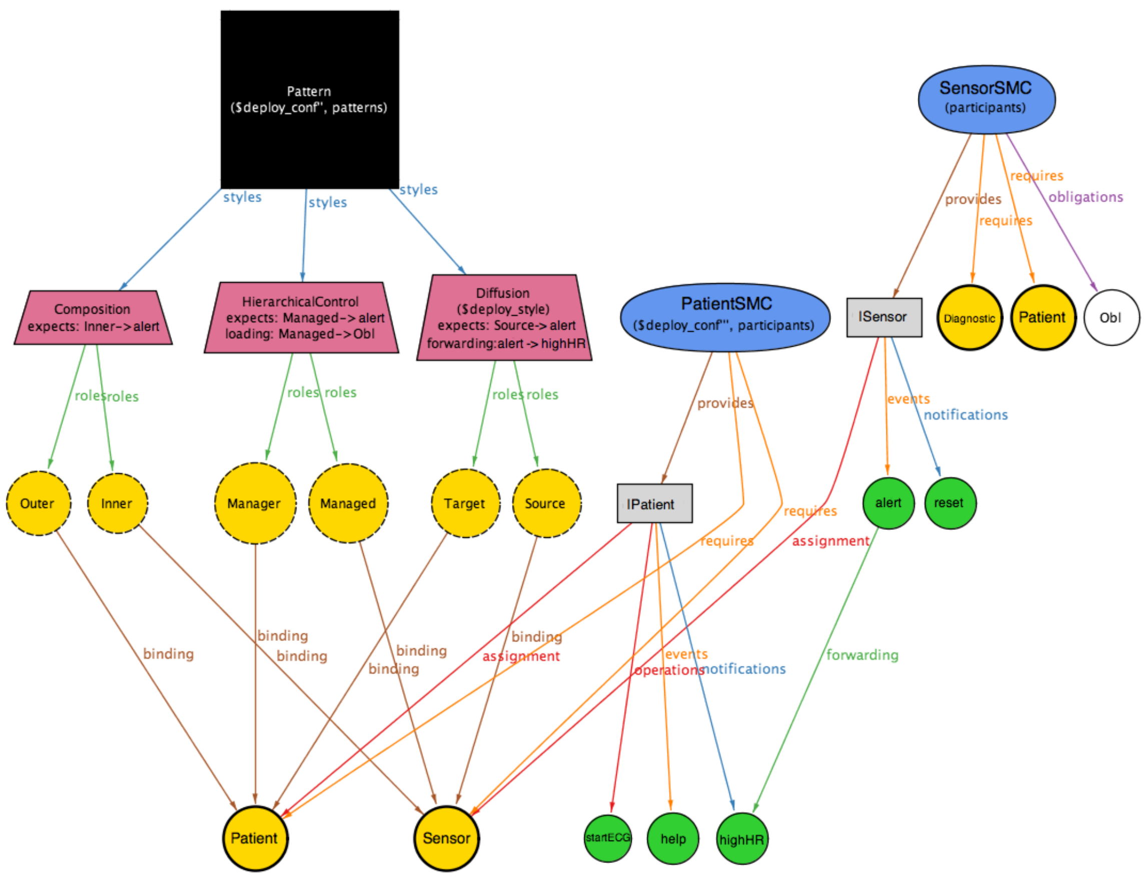


Figure 5.16: Alloy graphical representation of an architectural configuration

This example in particular illustrates the deployment of the *Diffusion* style (the style is labelled as *"$deploy_style"*). This style 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 style-specific role *Source* to the domain 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 style. Similarly, this style is also labelled as *"forwarding: alert → highHR"* (this corresponds to the behaviour that must be added to the interaction after the style is deployed). 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*, which is in turn bound to the style-specific role *Source*) and the *highHR* notification (provided by *IPatient*, which is assigned to the role *Patient*, and bound to the style-specific role *Target*). The behaviour added by the deployment of other types of architectural styles can be shown in a similar manner, in the form of mapping of interfaces, forwarding of events or loading of policies.

This model can then be used to check, for example, whether all the SMCs enforcing policies have the required events forwarded to them (as these events are required for triggering the policies) as well as whether all SMCs have access to the interfaces required for enforcing the actions prescribed by policies. Alternatively, the formal analyses of a specific SMC interaction may determine that a valid interaction is not achieved for a given set of SMCs and policies, and that additional abstractions must be added to the interaction, e.g. an event must be forwarded or additional interfaces must be exchanged. These are discussed in the next section.

5.2 Model Analysis

Formalising SMC behaviour enables reasoning about interactions, to verify whether they are correctly specified and deployed and whether they achieve their intended behaviour. This increases confidence in the robustness of policy-based SMC interactions, as we are able to analyse them rigorously prior to instantiation and deployment in actual devices. Based on the types of inconsistencies described in Section 4.3.3 (with respect to bindings, assignments and policy deployments), we discuss here various types of verifications we are able to perform automatically.

We distinguish between:

- ***Static checks***: properties that must hold for any instance of the model, and must never produce a counter-example provided the semantics of the model was specified correctly, e.g. an SMC that is already composed should never be allowed to participate in any interaction outside that composition according to the rule of encapsulation. Such verifications are specified in the form of *assertions*, that can be checked through an exhaustive search for a given (finite) number of instances for each one of the signatures defined in the model, i.e. considering a given number or SMCs, roles, interfaces, architectural styles, etc; and

- ***Dynamic checks***: verifications that represent properties we want to test for a specific instance of the model, e.g. after a task loading style is deployed we can verify whether all SMCs enforcing policies have the required events forwarded to them, or whether further abstractions must be added. These verifications are usually specified as *predicates*, which define a property which must be satisfied by a particular instance of the interaction, i.e. an instance of *Configuration* or *ArchitecturalConfiguration*;

The list below is not exhaustive, but it exemplifies the types of analysis we are able to perform using the formal model specified in this chapter:

- Check whether the role requirements are satisfied by the SMCs assigned to these roles;
- Consistent policy deployment where all obligation policies have a corresponding positive authorisation, and no modality conflicts occur;
- Roles bound in a composition have no interactions with other roles outside that composition.
- Ensure a given SMC is bound to a particular abstraction, e.g. *sensor* SMCs should always be encapsulated by a *patient* SMC;
- An event-forwarding style can only be bound if a structural style is already bound ensuring the interfaces required for event forwarding were already exchanged;
- All policies have access to the events required for triggering these policies (either the SMC enforcing the policy provides the event or it is forwarded from a remote SMC);

- All policies have access to the interfaces required for action invocation (either the target SMC provides the action or it is mapped from another SMC).

We can further check for application-specific properties of an SMC's behaviour, whether in given circumstances authorisations permit the execution of actions that would threaten the integrity of the SMC or whether failure or departure of devices leaves the SMC in a state where it is no longer able to fulfill its primary functions. This is particularly important when developing body-area networks for health monitoring as the integrity of the sensor configuration influences the medical interpretation of the physiological parameters collected.

Below, we describe in more detail a number of dynamic verifications that can be performed using the formal model, namely for checking role assignments, policy deployment, and consistent combination of architectural styles.

5.2.1 Role Requirement

When architectural styles are bound to the roles required by an SMC, the requirements of each style become automatically associated with these roles. Thus, the roles required by an SMC will combine the requirements of all styles bound to them. This requires verifying that whichever SMC is assigned to one of these roles, its interface satisfies *all* the requirements associated with that role.

```
pred verifyRoleRequirements [conf: ArchitecturalConfiguration]
{
  all pat: conf.patterns, sty: conf.deployed, styrol: sty.roles {
    !(no styrol.(sty.expects)) ⇒ one itf: conf.participants.provides {
            ((itf → styrol.(conf.binding)) in conf.assignment) and
            (styrol.(sty.expects) in (itf.operations
                                      + itf.events
                                      + itf.notifications))
    }
  }
}
```

Figure 5.17: Verification of role requirements

Figure 5.17 illustrates the *verifyRoleRequirements* predicate used to verify this condition. It states that for the specific configuration given as argument, for all styles deployed and for all style-specific roles defined for these styles, the following must hold: if a style-specific role defines a non-empty set of requirements, then there must exist an interface provided by one of the participants such that this interface is assigned to the role bound to the style-specific role in question, and all the requirements associated with this style-specific role are satisfied by the union of such interface's operations, events and notifications. This property ensures that the SMCs involved in this interaction are capable of satisfying the requirements associated with their roles.

5.2.2 Policy Deployment

The formal specification also facilitates the checking for conflicting or inconsistent policy deployment across SMCs. Two types of checks are discussed in the following: the verification of *(a)* whether all obligations enforced by collaborating SMCs have a corresponding authorisation policy, and *(b)* whether the policies enforced by a set of SMCs are free from modality conflicts. More traditional types of policy analysis, such as application-specific conflicts [LS99] can be defined in a similar manner, e.g. nurses should never be authorised to administer medication on themselves.

Figure 5.18 illustrates the *noUnauthorisedObligation* predicate which determines whether all obligations have a matching positive authorisation. It states that for any SMC *smc1*, and for all active obligation policies, if *smc1* is enforcing this policy, then there must be an active positive authorisation policy, enforced by an SMC *smc2* which specifies the same action as in the obligation, and which has the same SMCs assigned to the subject and target roles in both policies.

The SMC interaction previously illustrated in Figure 5.10 shows an example of a valid policy interaction between SMCs, because each obligation policy has a corresponding authorisation policy that allows the action specified by the

```
pred noUnauthorisedObligation [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 some smcsubj: conf.participants |
           (((obl.subject).~(conf.assignment) in smcsubj.provides)
           and
           ((aut.subject).~(conf.assignment)) in smcsubj.provides)
         and some smctarg: conf.participants |
           (((obl.target).~(conf.assignment) in smctarg.provides)
           and
           ((aut.target).~(conf.assignment)) in smctarg.provides)
       }
 }
}
```

Figure 5.18: Verification of whether obligations have a matching authorisation

obligation to be executed in the target SMC. The assignments presented in Figure 5.10 ensure that the SMCs assigned to subject and target roles in an obligation are also assigned to subject/target roles in a matching positive authorisation policy.

Another property that can be easily checked for a given interaction is the occurrence of modality conflicts. Figure 5.19 illustrates the *noModalityConflict* predicate which determines whether an interaction between SMCs is free from modality conflicts. It states that for the configuration given as parameter, no two active authorisation policies exist in any SMC such that the subjects, targets and actions of the policies overlap, but their modalities are opposite.

Figure 5.20 illustrates an example of a modality conflict between SMCs. *DoctorSMC* has an obligation policy *Obl*, which defines *Doctor* and *Patient* as (respectively) the subject and target of the policy, the event *highHR* as the event required for triggering the policy, and the action *startECG* as the action to be executed when the policy is triggered. *Doctor* and *Patient* are roles required by this SMC. *PatientSMC* has two authorisation policies, *Aut0* and *Aut1*, which define *Doctor* and *Patient* as (respectively) the subject and target in both policies,

```
pred noModalityConflict [conf: Configuration]
{
  all smc: conf.participants {
    no disj aut1, aut2: (conf.active & ConcreteAuthorisation) {
      (aut1 + aut2) in (smc.authorisations)
      and (aut1.action == aut2.action)
      and (aut1.modality != aut2.modality)
      and some smcsubj: conf.participants |
          (((aut1.subject).~(conf.assignment) in smcsubj.provides)
          and
          ((aut2.subject).~(conf.assignment)) in smcsubj.provides)
      and some smctarg: conf.participants |
          (((aut1.target).~(conf.assignment) in smctarg.provides)
          and
          ((aut2.target).~(conf.assignment)) in smctarg.provides)
    }
  }
}
```

Figure 5.19: Verification of modality conflicts

with respect to the action *startECG*. However, while *Aut0* is a positive authorisation (labelled *"modality: Positive"*), *Aut1* is a negative authorisation (labelled *"modality: Negative"*). In terms of assignments, *DoctorSMC* provides interface *IDoctor*, which is assigned to the subject role (*Doctor*) of its obligation policy, and the same interface *IDoctor* is also assigned to the role *Doctor* required by the remote *PatientSMC* (*Doctor* is the subject role of both authorisation policies

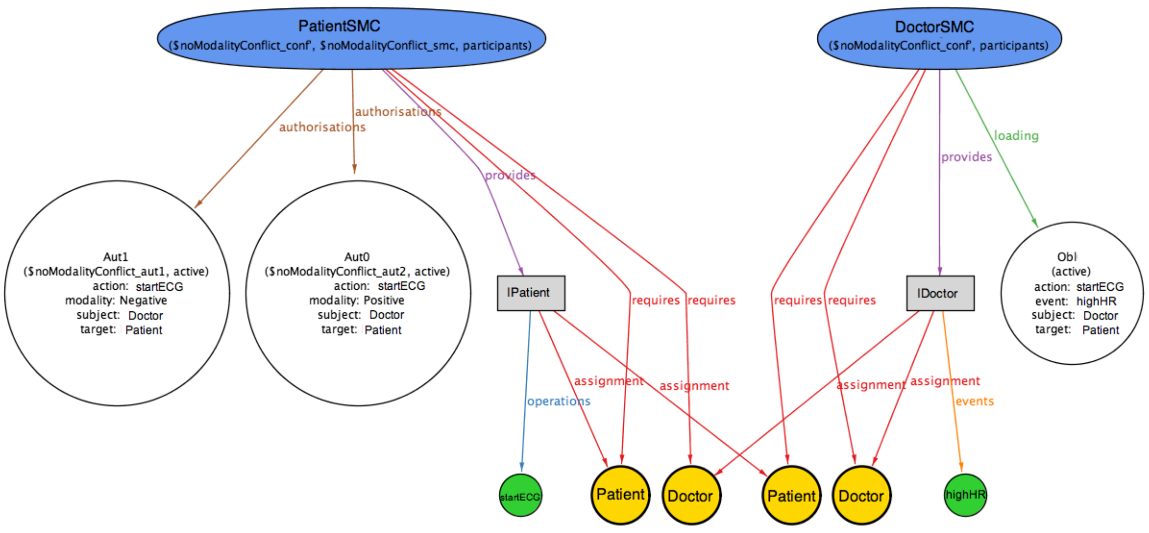


Figure 5.20: Invalid policy configuration between SMCs

enforced by that SMC). Similarly, *PatientSMC* provides interface *IPatient*, which is assigned to the target role (*Patient*) of both authorisation policies, and the same interface *IPatient* is also assigned to the role *Patient* required by the remote *DoctorSMC* (*Patient* is the target role of the obligation policy enforced by that SMC).

This is an example of an invalid policy configuration between SMCs because it contains a modality conflict. This happens because *Obl* (which is enforced by *DoctorSMC*) has two matching authorisation policies (which are enforced by *PatientSMC*): one (*Aut0*) that allows the execution of the action specified by the obligation policy, and another (*Aut1*) that denies the execution of the same action, i.e. the action *startECG* defined in the obligation policy is both allowed and forbidden to be executed by the authorisation policies enforced by *PatientSMC*.

5.2.3 Style Deployment

The formal specification also facilitates the checking of consistent deployment of architectural styles among SMCs. For example, we can check whether after the deployment of a task-loading style all SMCs enforcing policies have the required events forwarded to them, or whether additional abstractions must be included to the interaction. Similarly, it is possible to check whether these policies have access to all the required interfaces, as these are necessary for validating the remote actions prescribed by the policies.

Figure 5.21 illustrates the *verifyNoPolicyWithoutEvent* predicate which checks whether all SMCs enforcing obligation policies have access to the events required for triggering these policies. It checks, given a configuration, whether any SMC enforcing any obligation policy loaded by a *task-allocation* style either *(a)* provides the event required for policy triggering through one of its own interfaces or *(b)* receives the event via an event forwarding style already deployed in the same configuration.

Figure 5.22 exemplifies part of a configuration which does *not* satisfy this prop-

```
pred verifyNoPolicyWithoutEvent [conf: ArchitecturalConfiguration]
{
  all pat: conf.patterns,
      sty: (pat.styles & conf.deployed & TaskAllocationStyle) {
    let policySet = sty.roles.(sty.loading) {
      all pol: policySet {
        all smc: pol.~(conf.loading) |
          (pol.event in (smc.provides.events)) or
         ((pol.event → smc.provides.notifications) in conf.forwarding)
      }
    }
  }
}
```

Figure 5.21: Predicate for checking whether an SMC either raises or receives the events required for triggering its obligation policies

erty. In this case, *DoctorSMC* was loaded with the obligation policy *Obl1*. The policy specifies that it is triggered by the event *highHR*. However, the interface provided by *DoctorSMC* does not support event *highHR*, nor is this event forwarded to the SMC. This means that because the event will never be raised in *DoctorSMC*, the policy will never be triggered.

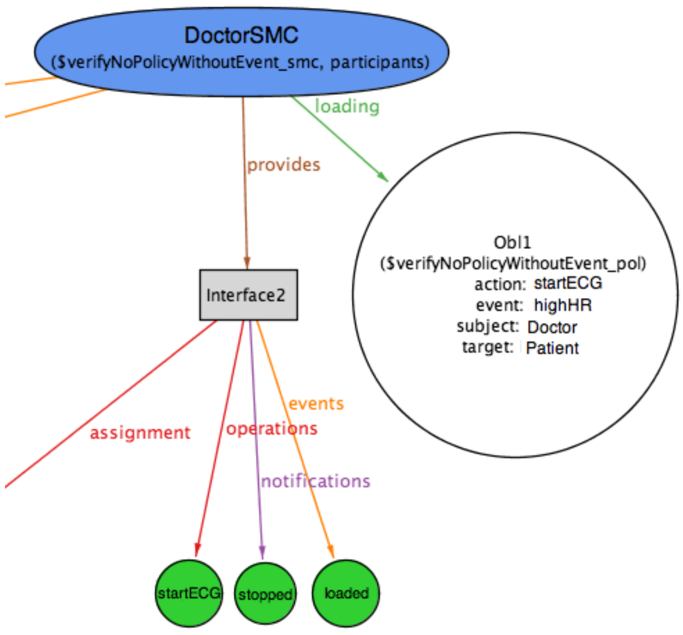


Figure 5.22: Example of an incorrect policy deployment

Similarly, we can define a predicate that verifies whether an SMC has access to the interfaces required for validating the actions prescribed by a policy. Figure 5.23 illustrates the *verifyNoPolicyWithoutAction* predicate. It checks, given a configuration, whether in any SMC enforcing any obligation policy loaded by a *task-allocation* style either *(a)* the policy's target provides the action required for policy triggering through one of its own interfaces or *(b)* this action is mapped to the policy's target's own interface via a structural style already deployed in the same configuration.

```
pred verifyNoPolicyWithoutAction [conf: ArchitecturalConfiguration]
{
  all pat: conf.patterns,
      sty: (pat.styles & conf.deployed & TaskAllocationStyle) {
  let policySet = sty.roles.(sty.loading) {
    all pol: policySet {
      all smc: pol.~(conf.loading) |
        pol.action in
          (pol.target & smc.requires).~(conf.assignment).operations or
        pol.action.^(conf.mapping) in
          (pol.target & smc.requires).~(conf.assignment).operations
      }
    }
  }
}
```

Figure 5.23: Predicate for checking whether an SMC has access to the actions prescribed by its obligation policies

These checks guarantee that interactions were specified correctly and that the SMCs involved are capable of enforcing their policies, before these interactions are implemented in physical devices. Manual implementation demands constant re-verification every time an interaction increases in size, and every time this happens it is more likely errors or inconsistencies will occur. In contrast, the predicates defined in the formal model can be used to verify automatically both smaller as well as more complex interactions, as they logically define what properties must hold for the interaction to be considered *correct*.

5.3 Discussion

Several studies have looked at the (conflict) analysis of policies in various forms [FKMT05, BLR03, CFP+06, RLSC+05], some of them based on model-checking techniques. Some of this work continues to date focusing on more complex policy languages and forms of analysis [JLT+08, CLM+09]. In particular, [BLR03, CLM+09] use *event calculus* to represent temporal enforcement of policies, showing how the fulfillment or violation of obligations affects the behaviour of the system and of other policies. In contrast to these studies our focus is not on the ability to detect policy conflicts, but to specify unambiguously the desired behaviour of interacting Self-Managed Cells and then verify if these SMCs are capable of enforcing their policies. The model specification presented in this chapter does not cater for the temporal enforcement of policies, i.e. we do not model a policy being triggered by an event. This type of analysis is possible but would require the formalisation of each of the managed object's behaviour in an SMC. Instead, this model focuses on the correct establishment of interactions and subsequent policy deployment.

Using this formal specification we can model specific policy-based SMC interactions and verify the correctness of these interactions before their actual implementation and deployment in physical devices. In this chapter a few examples of model verifications that can be performed were discussed: we can type-check role assignment across distributed SMCs ensuring that the SMCs involved in the interaction support the requirements for their respective roles, we can detect omissions from the configurations e.g. obligation policies that do not have a corresponding authorisation policy, and we can analyse consistent architectural style deployment where the exchange of policies must be accompanied by adequate exchanges of events and interfaces. Similarly, it is possible to check for application-specific properties of an SMC's behaviour which are violated due to dynamic changes such as the loading of new policies, the failures of components or the addition of new ones.

The formal model presented in this chapter is used for design-time analysis of SMC interactions, in order to verify whether a given set of SMCs are capable of establishing specific policy-based collaborations which satisfy a number of properties. This analysis is typically performed before implementing the interactions and deploying the policies in physical devices. However, during run-time, resources may fail, SMCs may leave an interaction and other SMCs may join it. Therefore, support for the dynamic verification of SMC interactions is also needed. Even though the formal model presented in this chapter could also be used to perform run-time verifications, this would require re-evaluation of the predicates defined in Alloy when changes in the available resources within the SMC occur. This limitation is discussed in more detail in Chapter 8.

The next chapter describes the implementation aspects of SMC interactions and our prototype, and presents its evaluation.

Chapter 6

Implementation and Evaluation

This chapter discusses the implementation of the framework for specification and establishment of SMC interactions, and then describes its evaluation. The evaluation covers three different perspectives, in particular focusing on the *memory footprint* and *performance* of the implementation, and on the *functionality* it supports for developing real applications. While memory and performance will be discussed in this chapter, the functional evaluation is presented in the form of a case-study scenario which will be described in the next chapter. Before describing the implementation aspects of our prototype and its evaluation, we present a brief overview of the Ponder2 framework.

6.1 Ponder2 Framework

The implementation of the framework for SMC interactions was done in Java, relying on the infrastructure provided by the Ponder2[1] policy framework. Ponder2 comprises a general-purpose object management system. It implements a policy execution framework that supports the enforcement of both obligation and authorisation policies. Policies are written in terms of *managed objects* (*MOs*), which are stored in a local domain service which implements a hierarchical namespace.

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

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 and allows the creation of user-defined custom managed objects, e.g. *adapters* for interfacing with a *temperature sensor*. These managed objects are programmed in Java as well. Managed objects serve as *factories* for creating object instances (i.e. for a given application) which are subsequently stored within the local domain. 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. *PonderTalk*'s syntax is based on *Smalltalk*, in which messages can be sent to objects. A *PonderTalk* statement is defined as a reference to a managed object (possibly stored in the domain hierarchy), followed by zero or more messages to be sent to the object. A message may be a simple command or it may be parameterised. The example below illustrates an example *PonderTalk* statement:

$$root/myObject\ print:\ \text{``}Hello\ World\text{''}.$$

In this example, the message *"print:"* is sent to the object *"myObject"* which is stored in the *"root"* domain of the local Ponder2 instance. The message receives as argument the string *"Hello World"*. This example assumes that *"myObject"* accepts the *"print:"* message, otherwise an exception will be raised.

PonderTalk commands are *linked* to Java methods defined in the corresponding managed object by using Java *annotations* (i.e. @*Ponder2op()*). Thus, a *PonderTalk* command such as:

$$root/myObject\ addPolicyBehavior:\ \text{``}policy\text{''}\ \ to:\ \text{``}role\text{''}.$$

is linked to a Java implementation in the corresponding managed object by using a Java *annotation* of the form:

```
@Ponder2op("addPolicyBehavior : to : ")
public void p2_addPolicyBehaviorTo(P2Block policy, String roleName)
{
    //Implementation of the method comes here
    //...
}
```

This decouples the Java implementation of the methods from their invocation for the purposes of configuring and controlling a Ponder2 system. New *factory objects* can be loaded in a Ponder2 interpreter and objects can be created from these factories on demand, permitting commands to be sent to these objects dynamically via PonderTalk messages. This mechanism is of central importance in realising the SMC's control-loop and its adaptation strategy, as it allows new management policies to be created as required during run-time and actions to be invoked automatically in response to context changes that occur within each SMC. Further details about Ponder2 and *PonderTalk*'s language syntax can be found in the Ponder2 website (http://www.ponder2.net).

6.2 Prototype Implementation

A prototype was implemented in order to demonstrate the concepts presented in this thesis. The presentation of the prototype is divided into two parts: initially, we describe the implementation of the basic aspects for facilitating SMC interactions, which support the establishment of SMC collaborations and promotes task exchanges based on the roles to which these SMCs were assigned. We then discuss the implementation of a library of *architectural styles* for systematically building SMC collaborations, which supports a much more general model based on richer abstractions for the structure, management and communication aspects of an interaction.

6.2.1 Methodology Overview

Ponder2 supports the specification of the core concepts which are pertinent to the SMC model, such as *events* and *policies*. However, to realise the framework proposed in this thesis, a number of Ponder2 extensions were required. This section presents an overview of the main steps involved in specifying and enforcing SMC interactions, the main components in our implementation, and the runtime support required for our framework.

The *specification* and *establishment* of SMC interactions consists of:

1. Defining how a number of roles are expected to interact with each other via policy, event and interface exchanges, by reusing architectural styles and pre-defined patterns of interaction;
2. Verifying the consistency of this interaction specification according to a number of pre-defined logical properties by using the SMC formal model and the *Alloy Analyzer*;
3. Deploying the *interaction specification* in one or more devices running the *SMC runtime*. Each device will either:
   (a) Instantiate the interaction with other devices; or
   (b) Participate in an interaction instantiated by another device;
4. Repeat step 3 for any sub-patterns defined in the *interaction specification*.

Figure 6.1 presents an overview of the methodology for specifying and establishing SMC interactions. Initially, an *application designer* can rely on a number of abstractions for specifying an SMC collaboration for a given purpose, e.g. healthcare monitoring. This specification is defined in terms of a number of *roles* that define the placeholders for actual SMCs, a repository of *management policies* pertinent to a particular scenario, and a repository of *architectural styles* and *patterns* that provide reusable abstractions to define how policies are exchanged, and how the necessary exchanges of events and interfaces for policy enforcement are achieved.

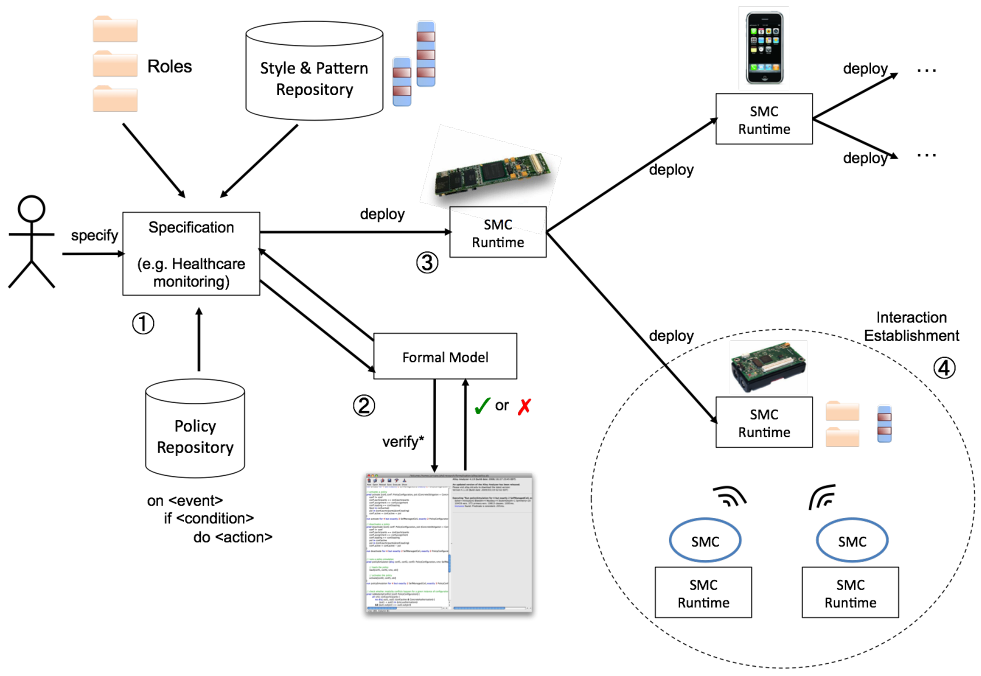


Figure 6.1: Specification and establishment of SMC interactions

The *interaction specification* is then given as input to the formal Alloy model for analysis, which can be used to verify a number of properties with respect to this specific interaction between SMCs. If the specification is successfully validated, it is ready for deployment in physical devices. Upon receiving an interaction specification, a device will instantiate the styles and patterns of interaction 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. 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 builds on the Ponder2 interpreter and adds to it a number of extensions required for facilitating SMC interactions (Figure 6.2). The standard functionality provided by Ponder2 implements *(a)* the *discovery service*, which permits the SMC to advertise itself to both devices and other

SMCs, *(b)* an *event bus*, which supports the underlying event-based infrastructure within the SMC, and *(c)* the *policy service* itself, which allows the specification and enforcement of both obligation and authorisation policies. The implementation of these services is presented in detail in [KDL$^+$07, K$^+$07]. These enable the basic functionality of the SMC as a feedback control-loop. Ponder2 also provides *(d)* a *command interpreter*, which allows *PonderTalk* commands to be sent to configure and control the Ponder2 system.

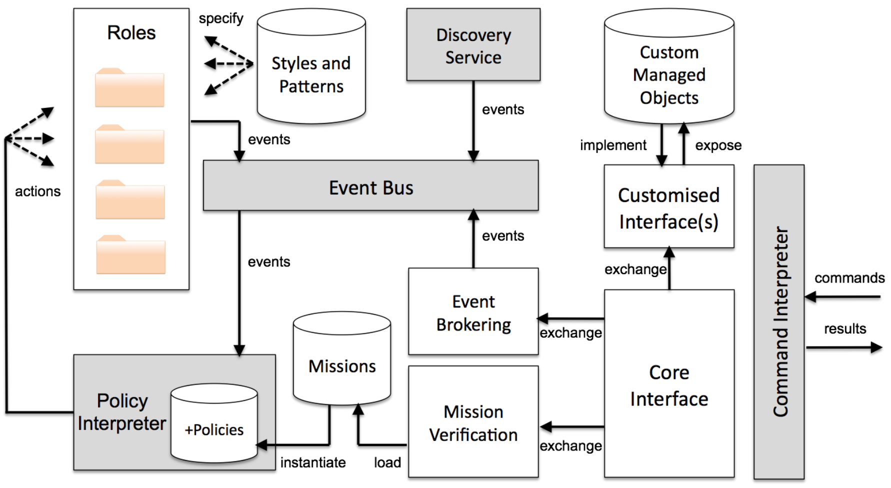


Figure 6.2: SMC runtime (shaded blocks are standard Ponder2 components)

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 specific managed objects that implement XML parsing of missions and their verification, and brokers that allow the subscription and forwarding of events between remote SMCs. A particular SMC can also have a dynamic set of application-specific *managed objects* (*MOs*) 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 styles enables the systematic specification of how a group of roles must interact. The implementation of these mechanisms is discussed in more detail in the following.

6.2.2 Basic SMC Interactions

As mentioned earlier, although the infrastructure provided by Ponder2 supports the core concepts which are pertinent to the SMC model, such as *events* and *policies*, several specialised managed objects had to be implemented in order to extend Ponder2 with the required functionality for supporting basic SMC interactions, e.g. *interfaces*, *roles*, *missions*. The implementation of these *managed objects* was done in Java, and these are used as factories for creating the specific object instances which are used by an application, e.g. a patient's *interface*, or an ECG monitoring *mission*. These are discussed in the following.

Roles and Policies

The *Role* managed object implements a placeholder within the local domain of the SMC, to which remote SMCs can be assigned. The role object defines an *expected interface* which specifies the requirements that remote SMCs must satisfy in order to be assigned to these roles, in terms of operations, events and notifications. This matching between the role's expected interface and an SMC's provided interface can be seen as matching the pieces in a jigsaw puzzle, where required roles must be fulfilled by the correct SMCs in order to form an entire application (Figure 6.3). In our implementation, the role's expected interface is syntactically matched against an SMC's provided interface, although a more flexible approach could define the use of ontologies for increasing the expressiveness in interface comparison.

Role extends Ponder2's *Domain*, and implements additional checks for verifying interface compatibility. Policies are then defined in terms of a number of roles, such as *"coordinator"*, *"authenticator"*, *"surveyor"*, etc. The policies will apply to the SMCs which are assigned to the respective roles, provided these SMCs'

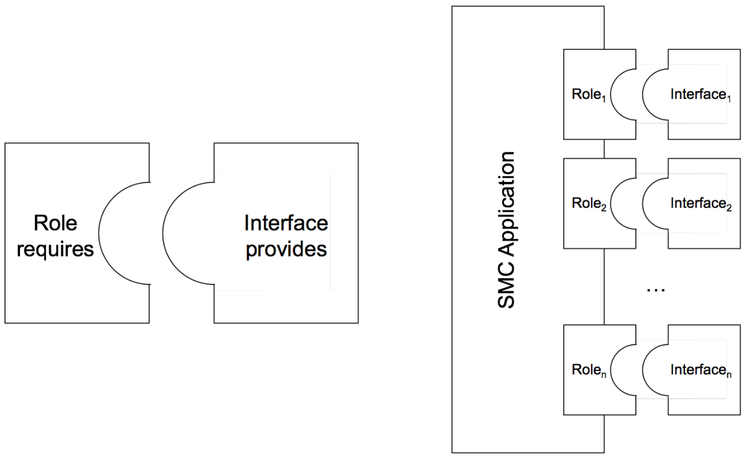


Figure 6.3: Roles used as placeholders for constructing SMC applications

interfaces support the requirements for their roles. This process is sketched in Figure 6.4.

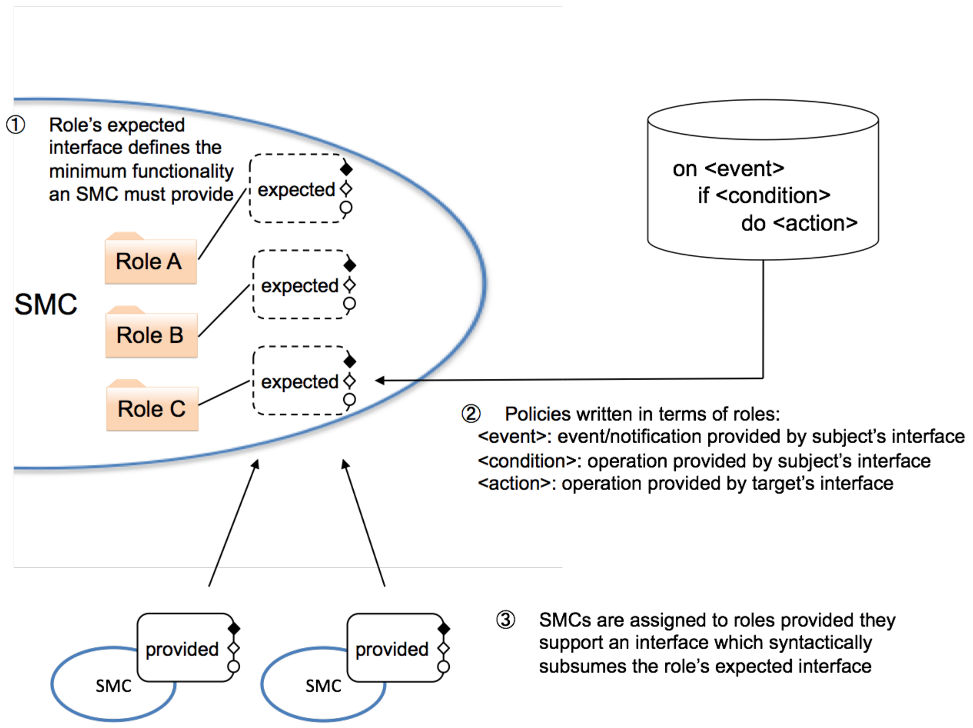

① Role's expected interface defines the minimum functionality an SMC must provide
SMC
Role A
Role B
Role C
expected
expected
expected
on <event>
if <condition>
do <action>
② Policies written in terms of roles:
<event>: event/notification provided by subject's interface
<condition>: operation provided by subject's interface
<action>: operation provided by target's interface
provided
SMC
provided
SMC
③ SMCs are assigned to roles provided they support an interface which syntactically subsumes the role's expected interface

Figure 6.4: Roles, expected interfaces and policies

Figure 6.5 illustrates an obligation policy being associated with the *"authenticator"* role (subject). The policy itself is defined within a *PonderTalk block*, which defines a section of one or more statements (within square brackets) whose execution can be delayed until it is decided that the block should be evaluated.

This allows policies to be defined during specification time, delaying their creation until they are loaded into an SMC. The snippet contained in the block in the example defines the creation of a policy from an *"ecapolicy"* factory. The policy specification is created and stored in the role, however the instantiation of the actual policy will be delayed until an SMC is assigned to the respective role during run-time. This policy specification was implemented to define the authentication of a discovered SMC, as described in the case-study presented in Section 3.6. The policy is triggered by an event of the type *"nodeReplied"*, which must be supported by the authenticator's interface. This in turn causes the action *"verifyCredentials"* to be executed locally on the authenticator[2] itself (target). In our implementation the SMC assigned to the authenticator role supports the operation *"verifyCredentials"* and provides an implementation for the validation of *X.509 digital certificates* through a custom managed object which uses the standard *java.security* package.

```
addPolicyBehavior:
  ([/policy
      at: "pol1"
      put: [obj := /factory/ecapolicy create.
            obj event: /event/nodeReplied.
            obj action: [:name :address :cap :credentials |
                 /roles/authenticator
                      verifyCredentials: name
                      from: address
                      capabilities: cap
                      credentials: credentials.].
            obj active: true.] value.
  ])
to: /roles/authenticator.
```

Figure 6.5: Authentication policy in PonderTalk

Similarly, Figure 6.6 illustrates an obligation policy associated with the *"coordinator"* role (subject). This policy specification was implemented to define the preferences for the assignment of SMCs to roles, as described in Section 3.6. The policy is triggered by an event of the type *"nodeAuthenticated"*, which must

[2]For the purposes of this implementation, mock *certification authorities* (CAs) were created, and *X.509 digital certificates* from these CAs were generated, using the *OpenSSL* package (http://www.openssl.org/). Certificates were then loaded into the SMCs that will join an interaction, whereas the SMC assigned to the authenticator role was parameterised with the relevant PKs for these certificates.

be supported by the coordinator's interface. The condition for policy triggering evaluates whether the string *"video"* is among the list of capabilities *"cap"* of the SMC indicated by the event. The list of capabilities supported by the SMC is obtained from one of the arguments of the event. If the condition is satisfied, the *"assign"* action is invoked in the *surveyor* role (target), which will cause the assignment of the SMC (whose *"name"* and *"address"* were also provided as parameters of the event) to the respective role.

```
addPolicyBehavior:
  ([/policy
      at: "pol2"
      put: [obj := /factory/ecapolicy create.
            obj event: /event/nodeAuthenticated.
            obj condition: [ :cap | "video" in cap].
            obj action: [ :name :address |
                 /roles/surveyor
                     assign: name
                     from: address].
            obj active: true.] value.
  ])
to: /roles/coordinator.
```

Figure 6.6: Role assignment policy in PonderTalk

Core and Customised Interfaces

Interfaces define the functionality supported by SMCs. Each SMC provides one or more interfaces to remote SMCs. The implementation of interfaces was divided into two managed objects:

- *CoreInterface*: defines a set of primitives that are common to all SMCs, independent of their application purposes and supports operations for interface exchange and binding, and interface mapping. It also provides operations for exchange and subscription of events, as well as operations required for the exchange of missions and installation of policies; and
- *CustomisedInterface*: extends *CoreInterface* and adds to it application-specific functionality, e.g. defines operations for reading sensor data, or for setting new thresholds.

The functionality provided by a *customised interface* depends on the local resources and application-specific services that the SMC implements. For example, an SMC whose interface specifies an operation for verification of public-key credentials must implement the application-specific *managed objects* that support this functionality. Similarly, an interface can define functions for reading measurements of a sensor provided the SMC implements the managed objects that physically communicate with the sensor device.

Mission Specification and Loading

The *Mission* managed object is parameterised with a set of roles, and implicitly their expected interfaces, and groups a set of obligation policies which can be loaded into a remote SMC. We implemented the checks discussed in Section 3.4, that must be performed at *mission specification* in the source SMC, and at *mission loading* in the target SMC. This is summarised in Figure 6.7.

At mission specification (left hand side of the flowchart), the mission is parsed and each policy written in terms of the roles defined in the mission is matched against the expected interfaces of the respective roles. Thus if policy *"p"* specifies that action *"a"* must be invoked on role *"r"*, the mission will only be validated if action *"a"* is defined in the expected interface of role *"r"*. Although matching is currently only based on syntactical equivalence, a more flexible approach could take advantage of subsumption and ontological reasoning to add a fine level of granularity and expressiveness for interface comparison. The source then specifies the argument values for the mission, both the address of actual SMCs for the role arguments as well as any application-specific argument used by the policies, e.g. thresholds, rates, etc. If the source has the required authorisations, the mission is sent to the target SMC.

At the target SMC, the mission specification is validated before its policies are instantiated (right hand side of the flowchart). The first verification checks the mission structure and whether the specification contains only obligation policies written in terms of the roles which are part of the mission. Any attempt to load additional embedded code inside the mission, or invoking operations on

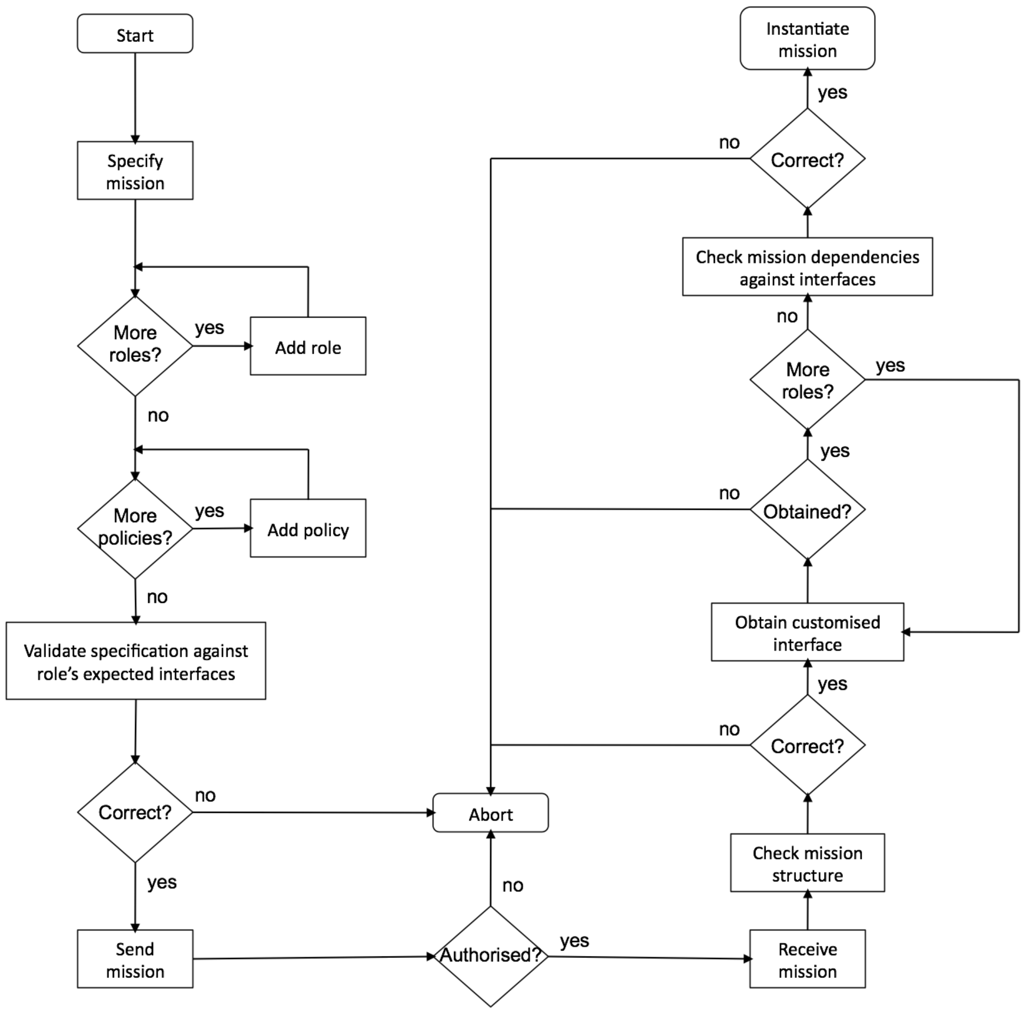


Figure 6.7: Mission specification and loading flowchart

objects other than mission roles causes the mission loading to be aborted. For each one of the roles, a customised interface is obtained for interacting with the respective SMC by contacting the SMC directly using the address provided by the source. Finally, the mission specification is matched against the interfaces of the respective roles. This verification can be against either the expected interface for each of the roles or against the customised interface of the actual SMCs assigned to these roles. In our implementation, we have chosen the latter option as it enables the mission to take advantage of operations provided by the specific SMC that the target will be interacting with, e.g. the nurse in a specific GP clinic or ward may make available specialised operations in addition to the standard functionality defined in the nurse's expected interface.

6.2.3 Architectural Styles and Patterns

More general interactions can be defined through the use of architectural styles. Interactions based on architectural styles allow the systematic construction of SMC collaborations which rely on much richer abstractions for their structure, task-allocation and communication aspects. A library of architectural styles was implemented in Ponder2 to show how styles can be rapidly and flexibly instantiated for building SMC interactions. Each style can be arranged independently in patterns of interactions, as the structure of the interaction should be defined independently from the way tasks are allocated and events are forwarded. We implemented managed objects to support the specification of various of the architectural styles presented in the catalogue in Chapter 4, and also a library where they can be chosen for constructing SMC interactions.

Style Implementation

The implementation of architectural styles is inspired by a technique for implementing layered object-oriented design known as *mixin layers* [SB02, SB98]. Mixin layers are used to define templates that specify a *collaboration* between a set of *classes*. A collaboration, in turn, defines a set of related *roles*. To some extent, *collaborations* in the mixin layer model can be seen as architectural styles in the model for SMC interactions, in that in both cases an independent aspect of the collaboration is designed as an interaction between a group of roles. The work on mixin layers is presented from a programming language standpoint, but an overview of the approach is shown in Figure 6.8, where three different objects, A, B and C, each supporting multiple roles, are simultaneously participating in collaborations $c1$, $c2$ and $c3$. Each collaboration prescribes certain roles for the objects, and an object does not need to play a role in all the collaborations. Similarly, in our framework each architectural style defines a number of roles, and SMCs play different roles in multiple styles. The use of mixin layers to define distinct layers of functionality to a group of interacting objects is related to the manner in which *aspect-oriented programming* is used to change the programmed semantics of a group of collaborating

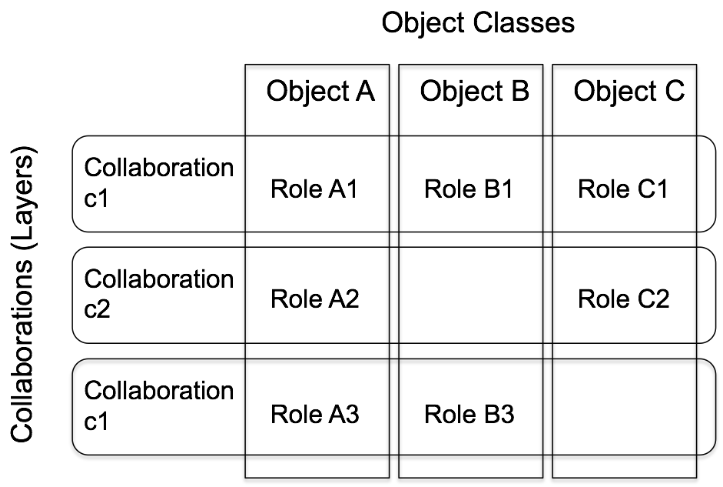


Figure 6.8: Mixin layers approach for interaction decomposition: collaborations are horizontal rectangles, object classes are vertical rectangles, and their intersection are roles within a collaboration

objects by defining a range of crosscutting concerns [KH01, KHH$^+$01].

An architectural style prescribes how a group of *interacting* roles must perform the exchanges of interfaces, events or policies. A set of domain roles within an SMC is said to be *interacting* if they are bound to style-specific roles within the same architectural style. Thus, architectural styles are applied on top of the roles defined in the SMC's domain and the respective interactions will be executed when actual SMCs are assigned to these roles, similar to what occurs when objects are assigned to roles in a collaboration in the *mixin layer* model.

In our implementation, each architectural style *managed object* functions as a *factory* for creating different parts of a large interaction, which defines a specific algorithm or protocol for the exchange of interfaces, events or policies. Figure 6.9 shows how a *composition* interaction is created from a *"Composition"* managed object in *PonderTalk*. Style-specific roles (*"outer"* and *"inner"*, in this case) defined in this managed object are bound to roles defined in the SMC's local domain (*"Patient"*, *"SensorHR"* and *"SensorTemp"*). Each architectural style is also parameterised with style-specific properties: an instance of the *"Composition"* managed object in this case is parameterised with *mappings* or *filterings* relevant to that application scenario. In this example, the oper-

```
comp := /factory/structural/composition create.
comp outer: "Patient".
comp inner: "SensorHR".
comp inner: "SensorTemp".
comp map:   "SensorHR.read"   to: "readHR".
comp map:   "SensorTemp.read" to: "readTempC".
comp map:   "SensorTemp.read" to: "readTempF"
                   filter: [ :readTempF | (1.8 * readTempF) + 32 ].
```

Figure 6.9: Code for the instantiation of a composition style between a *patient* and two *sensors*, written in Ponder2 syntax

ations *"SensorHR.read"* and *"SensorTemp.read"* are mapped to the operations *"readHR"* and *"readTempC"* exported by the interface of the SMC assigned to the *"Patient"* role. A filter is also applied to the operation *"SensorTemp.read"* (which converts the temperature readings from Celsius to Fahrenheit). The *filter* operation receives a *PonderTalk block* which defines the filter itself. Finally, the *encapsulation* is enforced in the composition style by stopping the inner SMCs from being discoverable. This will hide the sensors from external SMCs, keeping sensors as managed resources of the SMC assigned to the *"Patient"* role, but the selected operations will be mapped to the latter's interface.

The implementation of an architectural style *managed object* relies on the functionality supported by each participant's *core interface* to effect a specific algorithm or protocol that will achieve the exchange of policies, events or interfaces according to the abstraction defined by the style. The style managed object specifies what each style-specific role must execute in order to implement the style's semantics, and when actual SMCs are assigned to the respective roles the operations defined in the style are executed using the functionality defined in the SMCs's *core interfaces*. For example, the implementation of the *Diffusion* style requires installing an *event forwarder* at the *source* SMC, and installing the respective *event templates* at the *target* SMCs. Thus every time a specific event occurs inside the source SMC's event bus, this event will be automatically propagated to the target SMC's event bus, which will be able to handle such event because the necessary event templates were in place. This is achieved by using the operations *"p2_installEventForwarder"* and *"p2_installEvent"* available in the *core interface* of each SMC. The implementation of this style is illustrated in Figure 6.10.

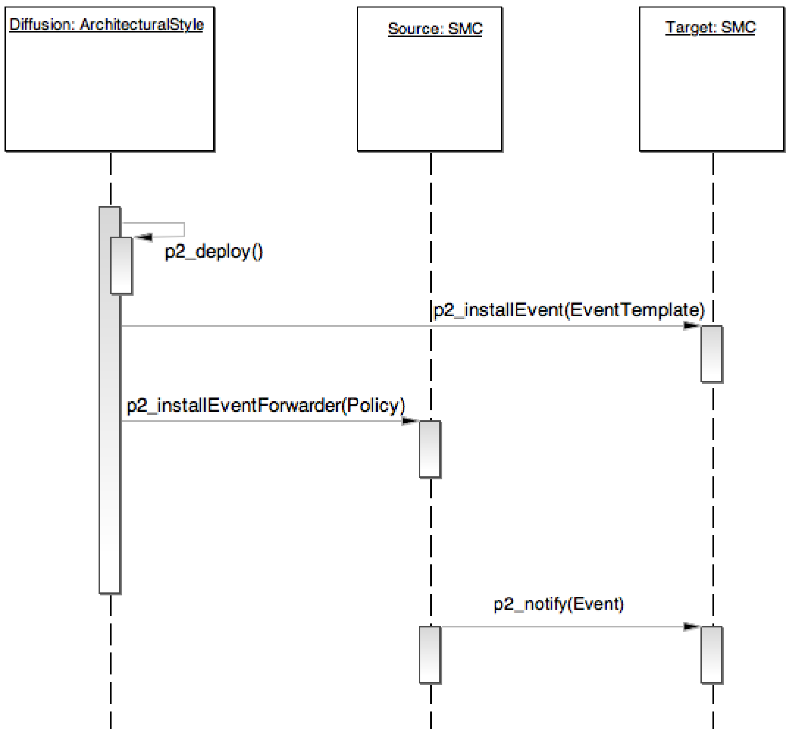

Diffusion: ArchitecturalStyle
Source: SMC
Target: SMC
p2_deploy()
p2_installEvent(EventTemplate)
p2_installEventForwarder(Policy)
p2_notify(Event)

Figure 6.10: Diffusion style implementation

Similarly, the implementation of the *SharedBus* uses the primitives defined in the *core interface* in a slightly more complex way in order to obtain the desired behaviour between multiple *publishers* and a centralised *blackboard*: a publisher must forward events raised within its event bus to the blackboard, which in turn must forward these events to all the other publishers. *Event templates* must be installed, so other SMCs are able to handle the events raised by a publisher, and each publisher must install an *event forwarder* to the blackboard, which in turn also installs an *event forwarder* to the publishers. The implementation of the *SharedBus* style is illustrated in Figure 6.11.

The behaviour of an architectural style (how the exchange is to be achieved) is defined only once, in the respective style's managed object, more specifically in the method *"p2_deploy"*. Each architectural style managed object defines how the different participants must behave in order to collaboratively enforce the

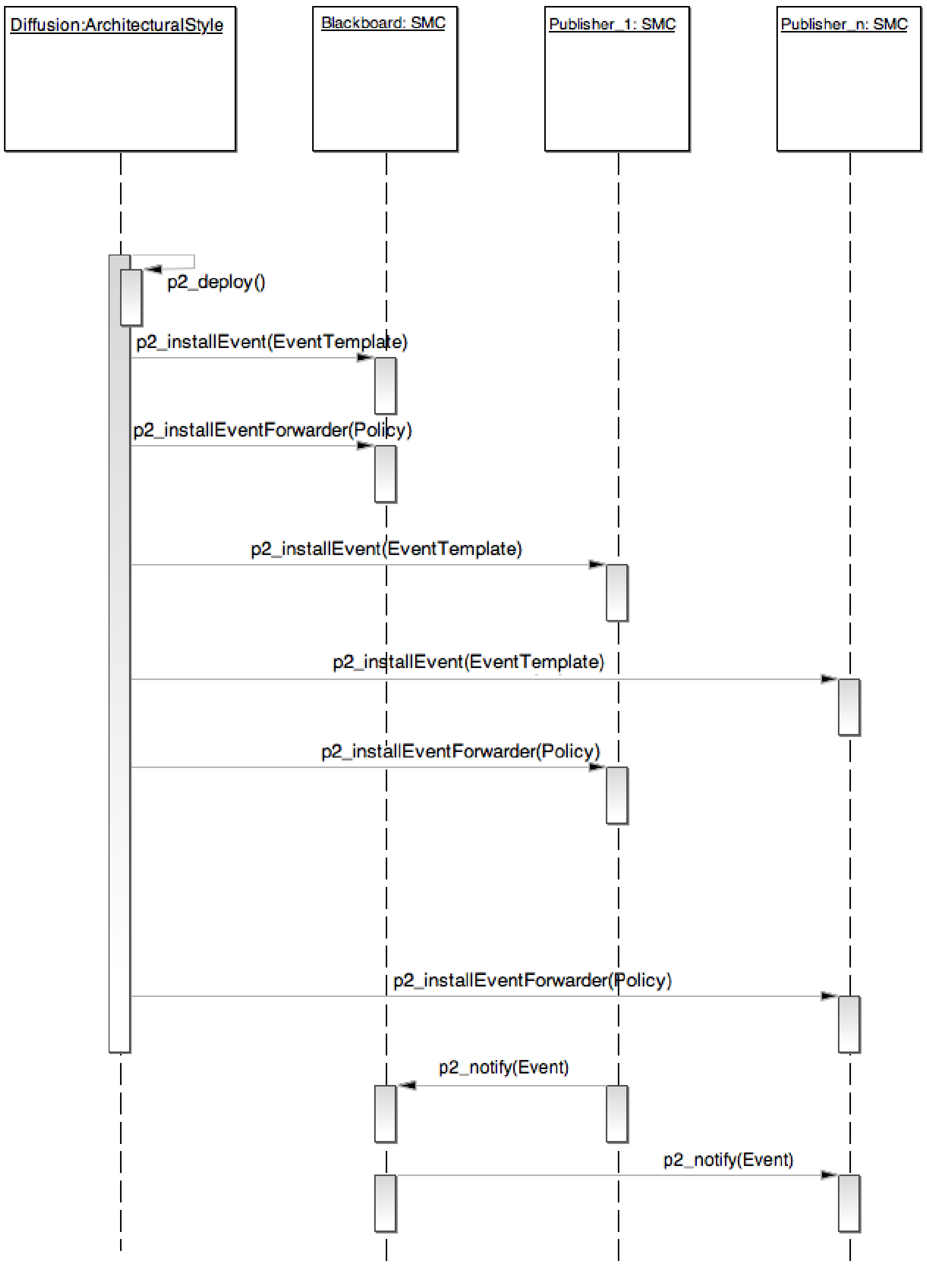

Diffusion:ArchitecturalStyle
Blackboard: SMC
Publisher_1: SMC
Publisher_n: SMC
p2_deploy()
p2_installEvent(EventTemplate)
p2_installEventForwarder(Policy)
p2_installEvent(EventTemplate)
p2_installEvent(EventTemplate)
p2_installEventForwarder(Policy)
p2_installEventForwarder(Policy)
p2_notify(Event)
p2_notify(Event)

Figure 6.11: Shared bus style implementation

semantics of the style. A given SMC does not need to implement all possible style managed objects, but only those which are relevant to the interactions in which the SMC participates. When an SMC is assigned to a domain role which is bound to an architectural style MO via a style-specific role, the fragment of Ponder2 code for this role (defined in the style MO) will be executed in the corresponding SMC. Note that loading additional managed objects or Java code into the participating SMCs is not a requirement for establishing a new interaction, as these SMCs may already have the relevant style MO definitions locally and then only execute the fragments of code defined for their style-specific roles. The *core interfaces* of the SMCs participating in the style are used for installing the different parts of the algorithm or protocol that defines the semantics of the style (Figure. 6.12). Thus each architectural style managed object can be seen as a *collaboration* between a group of interacting roles in the mixin layer model, which ensures that the semantics specified by a style is enforced in the involved SMCs.

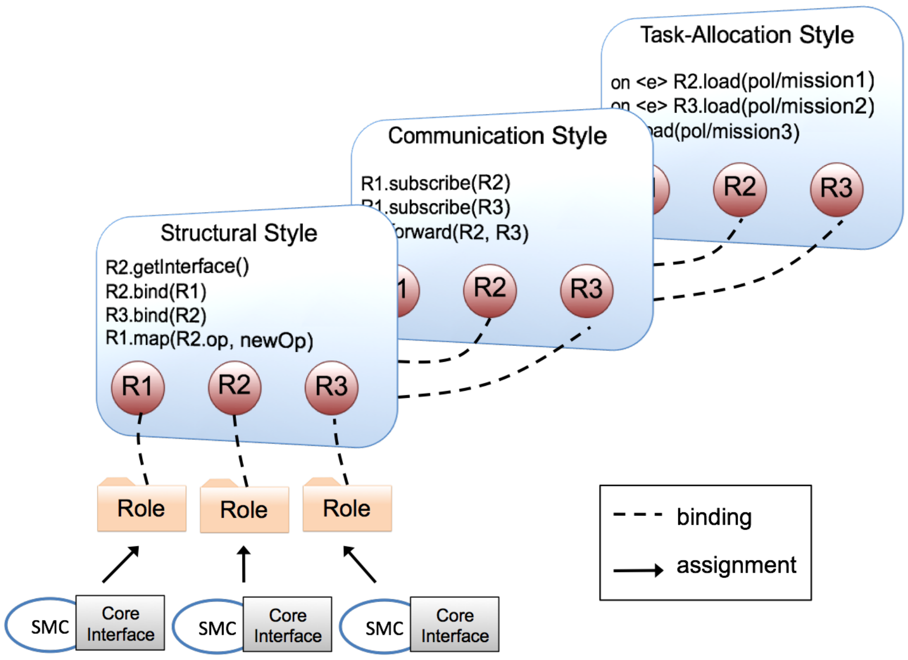


Figure 6.12: The implementation of each style is based on the *mixin layer* model and relies on the functionality provided by the core interface of the participant SMCs. These operations are executed when actual SMCs are assigned to roles

There are dependencies among architectural styles: structural styles must be deployed first, as they enable the exchange of *application interfaces*, e.g. *doctor* interface, *patient* interface; communication styles are then deployed to define patterns in terms of the events provided by these interfaces; task-allocation styles must be the last, as the policies loaded depend both on the operations provided by the application interface, as well as on the events forwarded by a

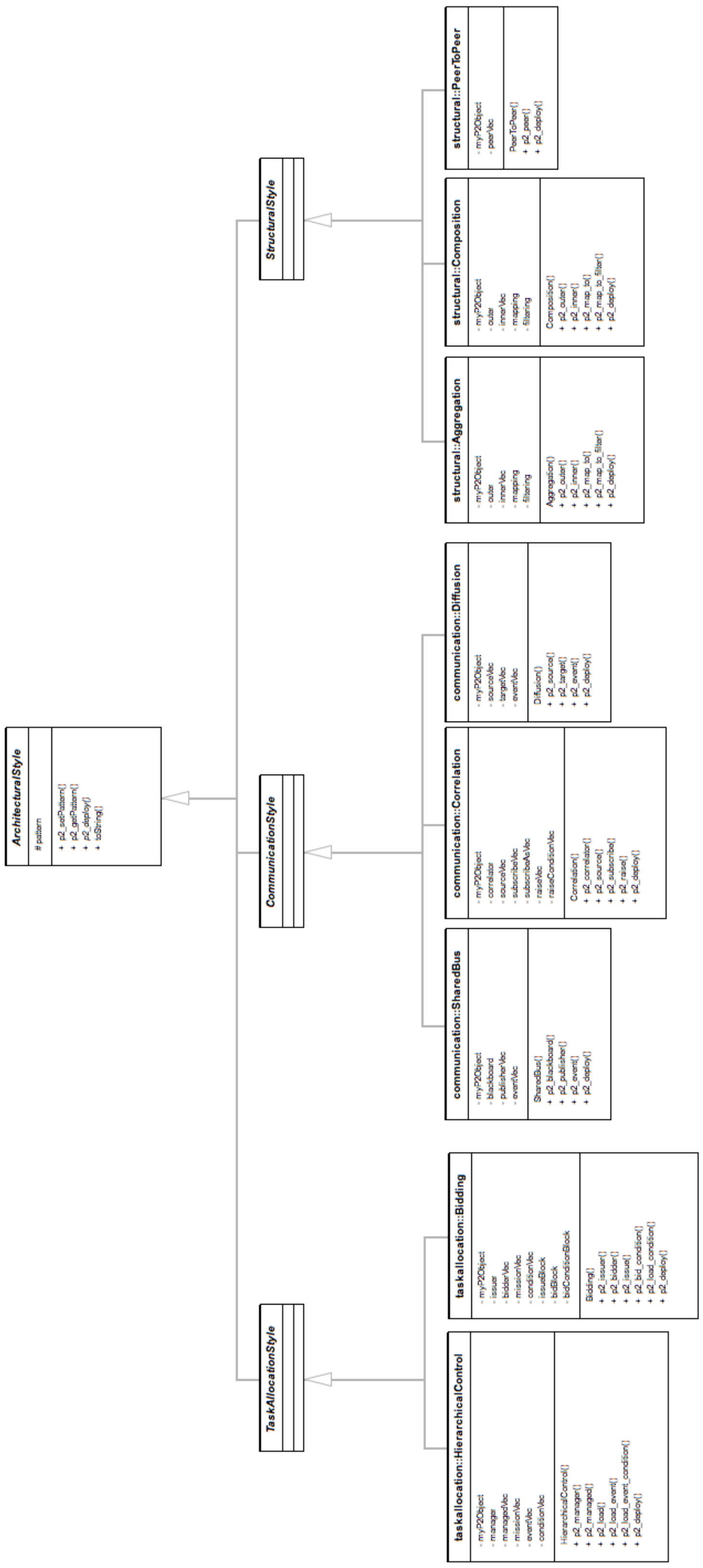


Figure 6.13: Architectural styles class diagram

communication style. Figure 6.13 illustrates a UML class diagram of the architectural style managed objects which were implemented in the core framework, i.e. the diagram excludes application-specific styles.

Patterns of Interaction

Patterns of interaction enable the specification and instantiation of interactions based on a specific combination of individual architectural styles. The *Pattern* managed object supports the specification of a set of roles and how these roles are arranged by individually combining architectural styles or sub-patterns. The interaction is systematically defined by binding architectural styles to these roles. The actual SMCs which must be used in this interaction will be assigned to the respective roles at pattern instantiation.

Figure 6.14 illustrates the creation of a *bodyarea* pattern which specifies the interactions between *"Patient"*, *"SensorHR"* and *"SensorTemp"* roles. The pattern relies on a *composition* structural relationship between the *"Patient"* (outer), and *"SensorHR"* and *"SensorTemp"* (inner) roles. The composition maps the

```
bodyarea := /factory/pattern
               create: "BodyArea"
               placeholders: #("Patient" "SensorHR" "SensorTemp").
bodyarea bind:
    [  comp := /factory/style/structural/composition create.
       comp outer: "Patient".
       comp inner: "SensorHR".
       comp inner: "SensorTemp".
       comp map:  "SensorHR.read" to: "readHR".
    ].
bodyarea bind:
    [  corr := /factory/style/communication/correlation create.
       corr correlator: "Patient".
       corr source: "SensorHR".
       corr source: "SensorTemp".
       corr subscribe: "SensorHR.HR" as: "HR".
       corr subscribe: "SensorTemp.temp" as: "temp".
       corr raise: "Patient.critical"
            condition: [:temp :HR | ((temp at: "value") > 40) &
                                    ((HR at: "value") > 150) ].
    ].
```

Figure 6.14: Body-area monitoring pattern in Ponder2 syntax

operation *"read"* from *"SensorHR"* to the patient's interface (*"readHR"*). The pattern in particular uses the *correlation* style to implement communication between the *"SensorHR"* and *"SensorTemp"* (source), and *"Patient"* (correlator) roles. In this example, heart rate and temperature events are used as parameters of a correlation function which raises the event *"critical"* in case the temperature readings are above 40 and heart rate readings are above 150. The correlation function parameterises the style and is flexibly defined as a *PonderTalk* block, which permits the specification of logical statements with the same degree of expressiveness of standard Ponder2 logical expressions.

The framework also supports the creation of more specialised architectural styles which can be included in the library for subsequent reuse. For example, a specialised correlator managed object can be created by extending the standard *correlator* architectural style managed object. A customised correlation function can then be defined in the subclass, which builds on top of the more general mechanism for event correlation provided by the super class. Similar types of customisation can be achieved by specialising the other styles. The extensibility of the framework is discussed in more details in Chapter 8.

6.3 Prototype Evaluation

The evaluation presented in this section focuses on two different aspects, namely, the *memory consumption* and the *performance* of the prototype implementation. These are discussed in the following.

6.3.1 Memory Consumption

This evaluation aims to show that the infrastructure for SMC interactions can be deployed in resources with limited computational power and memory, which are likely to be found in wireless ad-hoc networks involving smartphones, as well as unmanned vehicles and other small computing devices. Our prototype for supporting SMC interactions was deployed in two classes of lightweight, constrained devices: Gumstix[3] and Koala robots[4] (Figure 6.15). 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[5]. In the experiments we used JamVM version 1.4.5 and GNU Classpath version 0.91.

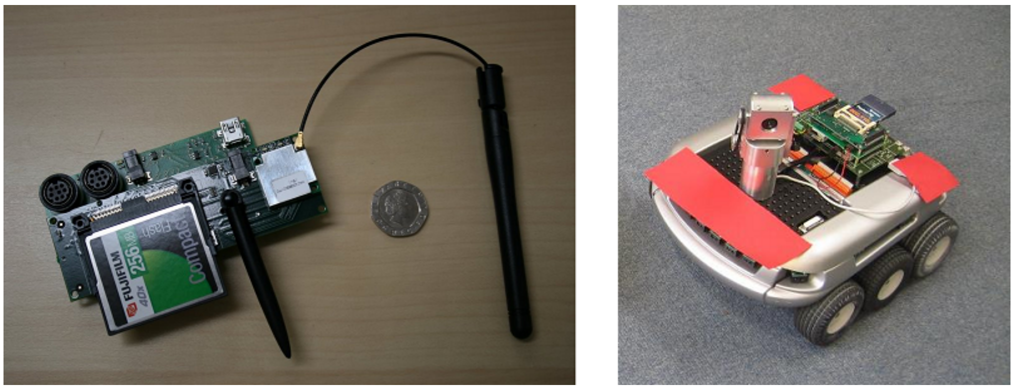


Figure 6.15: Gumstix (left) and Koala robots with video surveillance capabilities (right): each robot has 16 infrared proximity sensors around its body, and a video camera

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 running in a *"Coordinator"* role, and keeping the interaction specification and all the objects loaded in memory, required 15 MB for the Ponder2 process and 9,224 KB for the *rmiregistry* process[6] (*RMI* is one

[3]http://www.gumstix.com
[4]http://www.k-team.com
[5]http://jamvm.sourceforge.net
[6]By comparison, an empty JamVM and rmiregistry uses about 3,200 KB and 5,900 KB respec-

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

6.3.2 Performance

This section describes performance measurements of our prototype. The tests consisted in measuring the time taken for a Gumstix to assign a discovered Koala robot to a role, and then to load a variable number of policies. 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.

The graph in Figure 6.16 depicts our results. They 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, but as we increase the number of policies per role, this fixed cost tends to become negligible in comparison to the cost of loading and deploying policies (which increases linearly with the number of policies). 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

tively, and a JamVM running an empty Ponder2 instance and rmiregistry uses about 8,200 KB and 5,900 KB respectively.

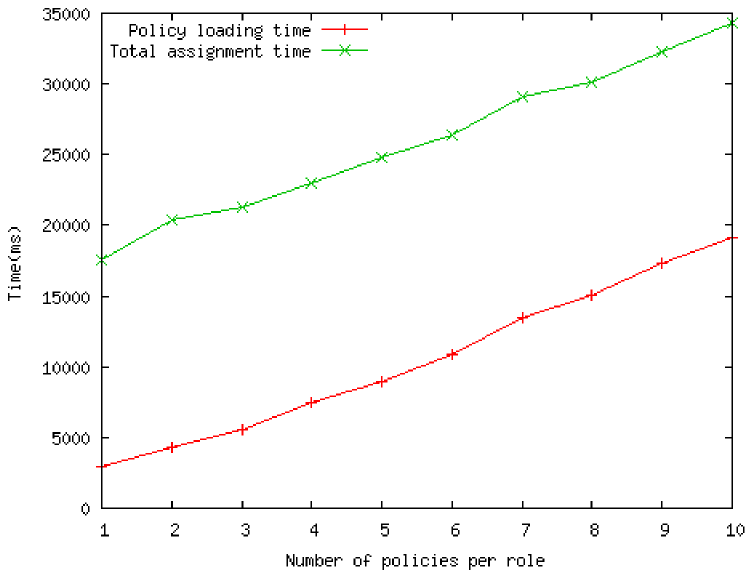


Figure 6.16: Total assignment time *versus* policy loading and deployment time

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.

6.4 Discussion

A prototype for SMC interactions was implemented in Java, which relies on the infrastructure provided by Ponder2. This chapter presented the implementation of basic SMC interactions, and then the more general use of architectural styles to enforce strategies of interface, event and policy exchanges. The use of architectural styles provides a more comprehensive model for specifying, instantiating and reusing interactions between SMCs. Architectural styles allow the rapid instantiation of different aspects of a collaboration, by supporting template interactions that enforce a specific algorithm or protocol for interface, event and policy exchanges. We deployed the prototype on constrained devices, such as Gumstix and Koala robots, in order to assess the suitability of the implementation in this class of resources. The memory footprint and performance results obtained in our experiments show that real policy-based SMC applications can run effectively in these devices.

The functional evaluation of our model consists of a case-study which will be presented in the next chapter. The case-study describes an application for the monitoring and treatment of diabetes, which relies on a number of SMCs and distributed policies to realise the desired collaboration between sensors, diagnostic devices and other resources. This case-study will better illustrate how the use of architectural styles facilitates the engineering of larger policy-based SMC collaborations, as well as interactions across collaborations, and the benefits of this approach.

Chapter 7

Case-Study: E-Health Monitoring

This chapter presents the use of the Self-Managed Cell framework in the design of an application for the monitoring and treatment of *diabetes mellitus*. There is an increasing interest in diabetes prevention and control, especially in the United Kingdom, due to the high percentage of the National Health budget spent on the treatment of this condition. The management and treatment of diabetes requires a complex combination of monitoring and drug delivery activities, which frequently depend on each other, e.g. extra physical activity will temporarily decrease insulin dosages. Cholesterol levels and cardiac monitoring also play an important role in the treatment of a patient with this condition.

In this chapter, we characterise the requirements for monitoring this condition and present how a system for diabetes monitoring and treatment can be designed through the composition and interaction of SMCs in a body-area and home monitoring set-up. These SMCs rely on policy-based interactions where adaptive actions are executed in response to measurements performed by on-body sensors in order to realise diabetes management.

Our focus is on the interactions between the SMCs that realise this scenario, rather than the detailed care protocol for diabetes management. Complex algorithms for the assessment of the patient's condition or similar functionality is assumed to be implemented as part of the application-specific managed objects. For example, we assume that an algorithm that classifies accelerometer input into activities is available.

7.1 Diabetes Mellitus

Diabetes mellitus is a syndrome of disordered metabolism often caused by a combination of hereditary and environmental factors, resulting in abnormally high blood glucose levels (a condition known as hyperglycemia) [UK09]. Blood glucose levels are controlled by the hormone insulin, which regulates the uptake of glucose from the blood into cells. Diabetes mellitus leads to high blood glucose levels due to defects in either insulin production (diabetes mellitus *type 1*) or insulin action (diabetes mellitus *type 2*) in the body (in this case the patient does not lack insulin but rather has resistance to its action).

Average glucose levels are around 80–120 mg/dl (or 4.4–6.6 mmol/l) before meals, and below 180 mg/dl (or 10 mmol/l) two hours after meals.[1] High glucose levels are considered to be above 288 mg/dl (or 16 mmol/l). Incorrect treatment of diabetes may lead to hypoglycemia, i.e. abnormally low blood glucose levels, below 72 mg/dl (or 4 mmol/l). Hypoglycemia may be caused by too much or incorrectly administered insulin; too much, unplanned or incorrectly timed exercises (physical activities decrease insulin requirements); or insufficient carbohydrate intake.

Medical conditions often associated with diabetes include high blood pressure and elevated cholesterol levels. Blood pressure levels for patients with diabetes should be kept below 130/80 mmHg.[2] This is achieved via drug therapy, and each added drug reduces blood pressure by 5–10 mmHg. This can also require extra monitoring and diagnosis to detect abnormal heart activity, e.g. by using an ECG monitoring device. Blood fat levels (total including cholesterol) should

[1]mg/dl is milligrams per decilitre, and mmol/l is millimoles per litre
[2]mmHg is millimetres of mercury

be kept below 72 mg/dl (or 4 mmol/l). Other illnesses and infections can raise blood glucose levels and require it to be tested more often.

Diabetes management depends on a complex combination of monitoring and treatment activities. In the next sections we present the requirements and the design of a policy-based application for the monitoring and treatment of diabetes using the *Self-Managed Cell* framework.

7.2 Requirements for Diabetes Monitoring and Treatment Application

We present the design of a body-area and home monitoring set-up that monitors context, patient activity and physiological parameters, and enforces drug delivery and data forwarding according to an initially pre-defined but adaptive collaboration between SMCs. Body sensors are used to keep track of the health condition of a patient, and to respond rapidly to abnormal values in the form of either on-body actuation, e.g. drug delivery, pacemaker activation, or the invocation of external services, e.g. requesting emergency assistance, data forwarding. We assume the use of implanted insulin delivery pumps that are activated in response to measurements made by blood glucose sensors. Blood pressure control can be achieved in a similar manner, relying on drug delivery pumps that react to monitored blood pressure levels.

An outline of the system's operation would typically be:

- Blood glucose levels are continuously monitored by glucose sensors, and these measurements are used for controlling the automatic administration of insulin via an insulin pump.
- Blood pressure levels are also monitored by sensors, and values are used for triggering the administration of drugs for hypertension control.
- The amount of daily physical activity the patient has undergone, e.g. running, walking, sitting, lying, is monitored using accelerometers. This is

used to adjust insulin delivery dosages, as physical activity decreases artificial insulin requirements.

- Diabetes is frequently associated with cardiovascular problems. Heart rate is monitored by sensors attached to the patient, e.g. a chest strap with embedded sensors, transmitting heart beat signals to a receiver, e.g. smartphone. Based on the signal transmitted, the smartphone can determine the current heart rate and proceed with further actions. For example, temporarily triggering an ECG (electrocardiogram) device available in the home environment, which can detect the waveform of heartbeats and record it, thus providing to the doctor relevant data on the heart condition of the patient, e.g. detecting repetitive patterns of abnormalities. ECG results typically include: *(a)* heart rate, *(b)* heart rhythm and *(c)* heart waveform.
- Data from the on-body sensors is collected and correlated on a smartphone, and transmitted to a local database in the home environment for synthesis and subsequent analysis.
- Continuous monitoring must allow data gathering and synthesis in the home environment, correlating glucose, cholesterol, physical activity and blood pressure levels for a period of weeks or even months.
- Subsequent delivery of synthesised data on a daily/weekly/fortnightly basis (depending on the patient risk condition) to an *electronic patient record* repository provided by the GP, e.g. via a web-service interface.
- The failure of any body sensor must be immediately reported to the patient's GP, requesting device replacement if necessary.

In the next sections we show how our requirements can be addressed by the design of an autonomous application that relies on policy-based interactions for performing adaptive actions. We are thus interested in adaptive autonomous behaviour and how the various devices and services required for realising this scenario are composed into more complex autonomous structures and interact with each other. We do not consider security aspects, e.g. privacy of patient

Table 7.1: Basic SMCs used in the diabetes scenario

Self-Managed Cell	*Description*
Patient	Patient's smartphone managing the body-area SMC
Surgery	Web-service SMC that automatically updates GP records
Doctor	Doctor's smartphone used for interacting and managing patient's devices
Glucose sensor	Sensor SMC for glucose monitoring
Blood pressure sensor	Sensor SMC for blood pressure monitoring
Accelerometer	Sensor SMC for activity monitoring
Cholesterol sensor	Sensor SMC for fat levels monitoring
Heart rate sensor	Sensor SMC for heart rate monitoring
Drug pump	Actuator SMC for drug injection and hypertension control
Insulin pump	Actuator SMC for insulin injection
Server	Local database for sensor data collection in the home environment
ECG device	Electrocardiogram diagnosis device
Alarm	Display for notifications in the home environment

records or cryptography of exchanged data, at this stage as work on these issues is ongoing elsewhere [KLS09, ZSLK09, Keo05, KLS04].

7.3 Application Outline

Based on the requirements described in the previous section, Table 7.1 lists the basic SMCs used in the construction of this scenario. These are divided into four groups of interacting SMCs:

- A personal SMC controlling a patient's *body-area network* for health monitoring typically runs on a smartphone or Gumstix[3] device hosting SMC management services that control several sensors such as glucose, blood pressure, cholesterol and acceleration hosted on BSNs[4] (Body Sensor Nodes). Actuators, such as an insulin pump or drug pump SMC, are employed and activated according to conditions monitored by the sensors.

[3]http://www.gumstix.com
[4]http://vip.doc.ic.ac.uk/bsn/

- A doctor or nurse SMC would typically interact with patients, loading monitoring tasks and collecting results. Monitoring tasks permit continuous observation of the patient's condition in his own home environment. Tasks loaded by the healthcare worker continually run on the patient's SMC, relying on information provided by his body sensors. In some circumstances, e.g. nurse visits or a regular visit to the GP, the patient also runs tasks for the re-calibration of his sensors, using devices owned by the healthcare worker.
- A home monitoring system collects data monitored by the patient's body-area network, correlating glucose, cholesterol, physical activity and blood pressure levels for a period of weeks or even months. When risky conditions are detected, e.g. hypoglycemia, alarms can be used to notify the patient that extra care must be taken, e.g. to take fast-acting carbohydrate such as glucose tablets – quantities will vary according to the situation, and instructions should be provided with the alarm. Data is temporarily stored locally for synthesis and identification of trends in the patient condition. Depending on the severity of the patient's condition, e.g. propensity to cardiovascular problems, ECG monitoring can be added to the monitoring set-up.
- A link between the patient's home monitoring system and the GP surgery is used to deliver periodically synthesised data to an *electronic patient record* repository provided by the GP. In an emergency, data is used to request immediate assistance, e.g. if the glucose levels reach critical thresholds, if the blood pressure is not effectively lowered by the administration of drugs or if an imminent heart attack is detected.

Figure 7.1 outlines the SMCs involved in this scenario and their interactions. Interaction ***(1)*** refers to the interaction between ***doctor and patient***, for the purposes of loading monitoring tasks or re-calibrating the patient's sensors. This is triggered when the patient visits the clinic, typically over a Bluetooth connection between the patient and doctor smartphones. Interaction ***(2)*** corresponds to the ***body-area network*** comprising the patient's smartphone and

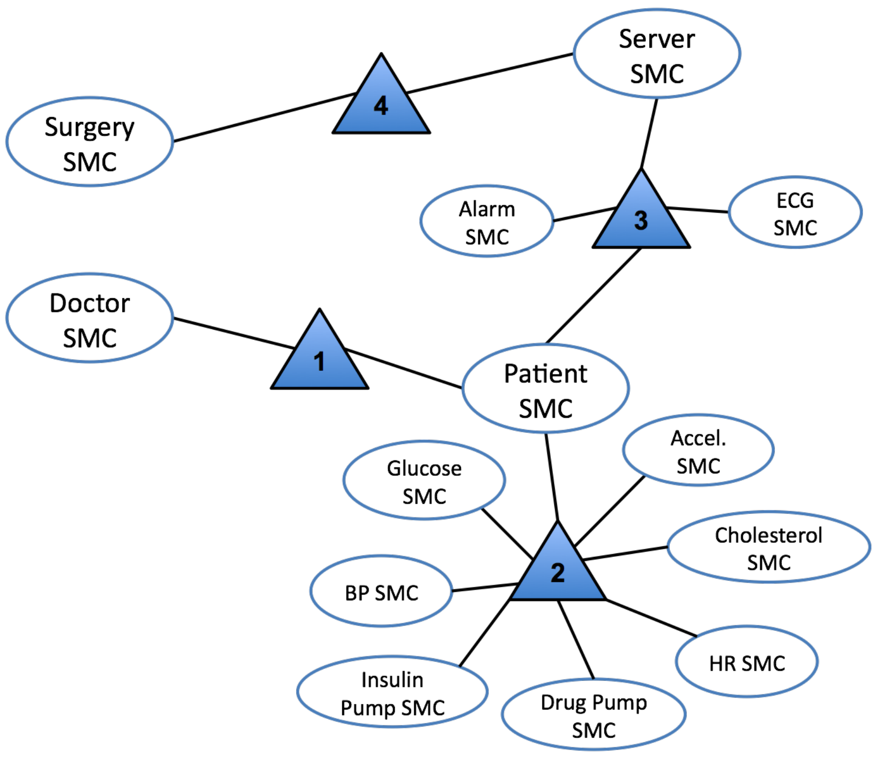


Figure 7.1: Diabetes monitoring scenario: each numbered triangle represents an interaction that requires a different combination of management abstractions.

the several sensors and actuators used for diabetes monitoring and treatment. Communication between these devices and the patient smartphone typically occurs through IEEE 802.15.4 radio links. Interaction ***(3)*** forms the ***home monitoring environment***, involving devices for data storage, alarms and external diagnostic available at the patient's home, interacting with the patient's own body-area network, often via a Wi-Fi connection between these resources. Finally, interaction ***(4)*** refers to the ***home environment and GP surgery*** interaction, through which the patient's records are updated or emergencies are reported. This interaction typically occurs via HTTP, considering the GP provides a web-service SMC for updating patient's records. Although the *Self-Managed Cell* concept provides a suitable abstraction for representing the autonomous components involved in this scenario, we still need adequate abstractions for expressing their interactions.

The next sections will elaborate these interactions using the principles presented in this thesis.

7.4 Patterns of Interaction

We can define patterns of interaction to be enforced by different SMCs in this collaboration. Patterns exhibit very specific management properties with respect to interface, task and event exchanges, which are dictated by the primitive styles that these patterns are composed from. In this section we present how two of these patterns are realised: a *body-area* pattern and a *home monitoring* pattern.

7.4.1 Body-Area Pattern

The arrangement of architectural styles presented in Figure 7.2 corresponds to a possible pattern for the body-area monitoring. It comprises the interactions between the device hosting the patient SMC and the sensors and actuators. This set of interactions (architectural styles) is typically established by the patient device. In the body-area network, the patient device and the several

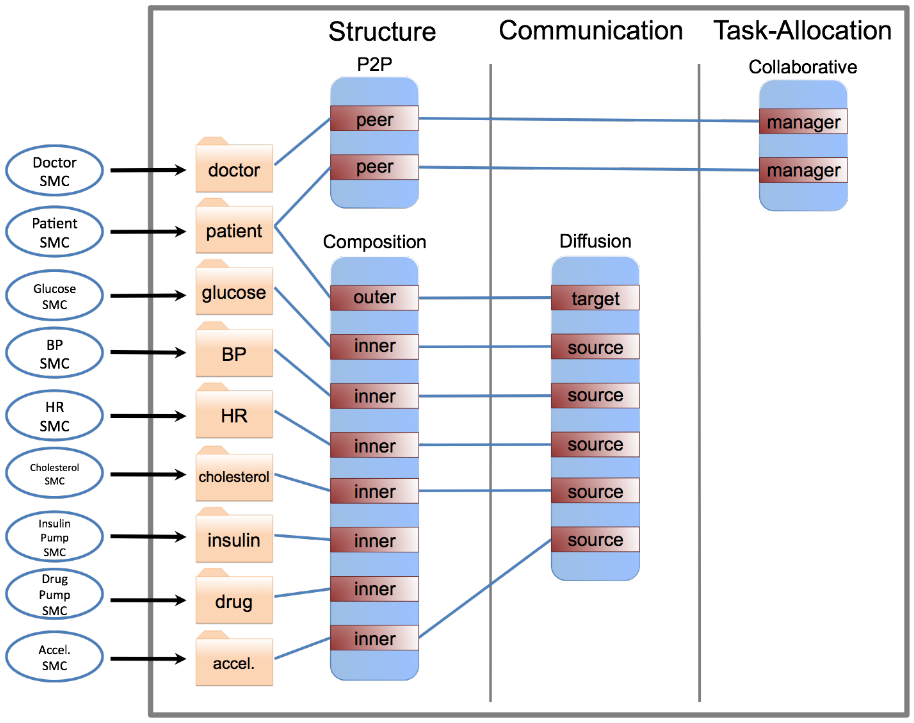


Figure 7.2: Body-area network pattern.

sensor and actuator SMCs are organised in a *composition* relationship as it requires *encapsulation* of the constituent devices. This is because we want the sensors to be visible and controlled solely by the patient device, and avoid sensors which belong to one body-area network to be discovered and interact with other external SMCs. Although the resources are encapsulated in the body-area network, operations that need to be accessed by the healthcare worker for reading sensor measurements or setting new thresholds are typically mapped and accessible from the patient's device interface. The body-area interaction typically relies on a simple event *diffusion style* from the sensors to the patient device. Normally, the sensors of a body-area network do not depend on each other's events – if that were the case, a shared bus or blackboard would be more suitable.

The pattern also comprises the interactions between the doctor and patient, especially for the purpose of the exchange of monitoring and re-calibration missions. This part of the interaction is typically enforced on the healthcare worker device, and can for example be triggered when the patient visits the GP surgery. A *peer* abstraction is more suitable for the interaction between patient and doctor, as a patient may interact with multiple doctors, and a doctor may interact with multiple patients. These interactions do not require mapping, filtering or encapsulation abstractions.

Based on the *functional requirements* identified in Section 7.2, the tasks that are required to be exchanged between the SMCs in this scenario are:

- ***Diabetes monitoring mission:*** this is the most important and also the most complex of the missions. It runs on the patient device and relies on the information provided by multiple sensors, e.g. glucose, blood pressure, cholesterol, heart-beat, accelerometer, to assess the patient condition and possibly to trigger adaptive actions, like the activation of insulin or drug pumps (some of these policies are illustrated in Figure 7.3).
- ***Sensor re-calibrating mission:*** is used by the patient for re-calibrating his sensors, using devices provided by the doctor or another healthcare worker. Sensor accuracy often depends on their calibration. Calibration

```
// insulin infusion policy
on glucose(level) do
    if level > 120 then
        /resources/insulinPump.dose(level - 120)

// potential hypoglycemic condition policy
on glucose(level) do
    if level >= 72 and level < 80 then
        /events/alert.raise("hypoglycemic condition " + level)

// confirmed hypoglycemic condition policy
on glucose(level) do
    if level < 72 then
        /event/alert.raise("hypoglycemic condition " + level)

// confirmed hyperglycemic condition policy
on glucose(level) do
    if level > 300 then
        /events/alert.raise("hyperglycemic condition " + level)

// blood pressure control policy
on bp(level) do
    if level > 130 then
        /resources/drugPump.dose(level - 130)

// physical activity reduces insulin requirements, and other
// policies must be activated
on context(activity) do
    if activity == "vigorous" then
        /policies/normal.disable();
        /policies/active.enable()
```

Figure 7.3: Example policies used in the diabetes monitoring mission.

```
// patient visits the GP and doctor re-calibrates the sensors
on energy(level) do
     /resources/patient/accel.set("sedentary", level <= 2);
     /resources/patient/accel.set("light",  2 < level <= 4);
     /resources/patient/accel.set("moderate", 4 < level <= 6)
     /resources/patient/accel.set("vigorous", level > 6)
```

Figure 7.4: Example policies used in the sensor re-calibration mission.

type and method is obviously dependent on the parameter being monitored by the sensor. For example, accelerometer-based activity monitoring can be used to measure the duration and frequency of activity undertaken under different intensity categories, such as sedentary, light, moderate and vigorous [PAVB02]. Calibration can be based on energy expended (kcal/min), using specialised equipment available when the patient visits the GP (Figure 7.4).

During patient visits to the GP, a *distributed control style* is typically established between doctor and patient SMCs which allows them to load missions mutually into each other. This allows a doctor SMC to load diabetes monitoring tasks into a patient SMC, but similarly a patient SMC also loads policies onto the doctor SMC to trigger re-calibration of the patient's sensors (the latter avoids the requirement for the doctor SMC to store calibration procedures for each individual patient).

7.4.2 Home Monitoring Pattern

The arrangement of architectural styles presented in Figure 7.5 corresponds to a possible pattern for the home environment monitoring. It comprises the interactions between the devices locally available in the home environment, e.g. local server, alarms, ECG diagnostic device, and the personal device hosting the patient SMC. These interactions are typically enforced by the local server, to

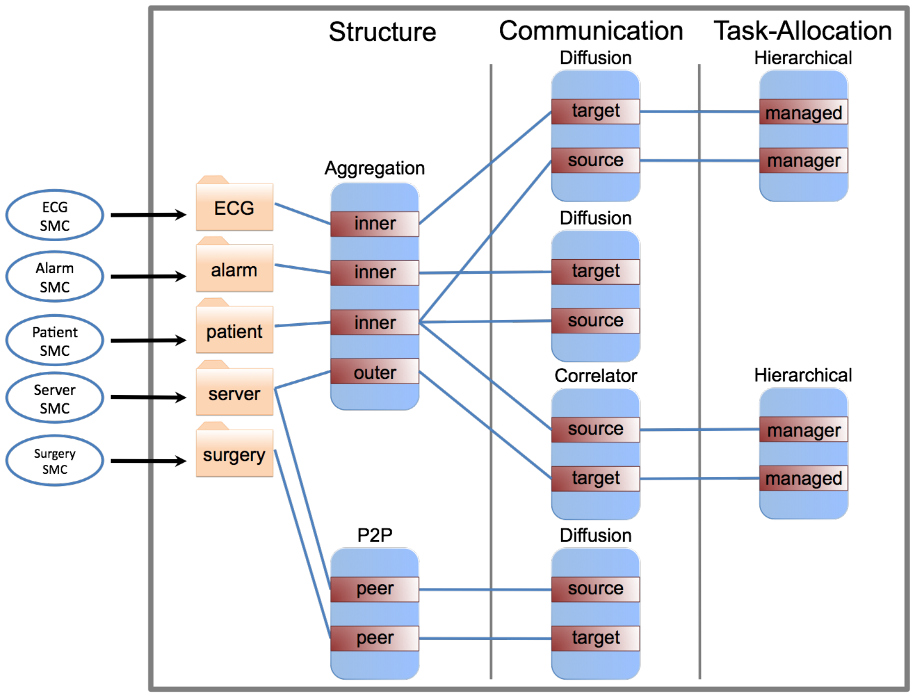


Figure 7.5: Home monitoring pattern.

control the interactions with other devices and appliances. The interaction in the home monitoring environment has an *aggregation* structure as it enables some degree of hierarchical organisation and filtering/mapping in the access to resources and appliances whilst still permitting the sharing of internal resources, e.g. the patient's body-area, with external SMCs that exist outside the aggregation. Event forwarding from the patient device into the home environment relies on a *diffusion style* to notify the occurrence of alarms, but also on a *correlator* to relay coarser grained information with respect to glucose or blood pressure monitoring – average levels over an hour, for example. Additionally, an event *diffusion* of heart rate events is required between patient and ECG devices.

This pattern also comprises the interactions between the home environment and the remote surgery, especially for the purpose of reporting the patient condition, which is also normally enforced by the home server. This interaction between the home environment and the GP follows a *peer* structure, similar to the interaction established between doctor and patient SMCs. For notifying an emergency, a *diffusion* of events between home environment and surgery is often required.

According to the *functional requirements* identified in Section 7.2, the tasks that must be exchanged between these SMCs are:

- ***ECG monitoring mission:*** in advanced stages of diabetes, the patient has a higher propensity to cardiovascular problems, and a doctor may recommend intensive monitoring and recording of the electrical activity of the heart, e.g. to anticipate the occurrence of a heart attack. This is achieved by a mission running on an ECG monitoring devices locally available in the home environment. A high heart rate triggers an ECG monitoring device, which can detect the waveform of heart-beats and provide to the doctor relevant data on the heart condition of the patient in his daily life. Such a device can temporarily record heart rate, heart rhythm and heart waveform, which can then be stored locally or sent to the GP (some of these policies are illustrated in Figure 7.6).

```
// activates the ECG monitor for a given period of time
on hr(rate) do
    if rate > 140
        /resources/ecg.start(30s)

// takes appropriate action depending on the results of
// an ECG monitoring
on ecgStopped(rate, waveform) do
    if rate > 140 and rate < 160 then
        /resources/localserver.store(rate, waveform);
        /events/alarm.raise(rate, waveform)

// takes appropriate action depending on the results of
// an ECG monitoring
on ecgStopped(rate, waveform) do
    if rate >= 160 then
        /resources/localserver.store(rate, waveform);
        /resources/surgery.callEmergency(rate, waveform)
```

Figure 7.6: Example policies used in the ECG monitoring mission.

- ***Data collection and synthesis mission:*** a data collection and synthesis mission is typically running on the home monitoring environment. It can be used for storing patient's data in a data collection hub available at the patient's home and for subsequently generating a synthesised summary of the collected data on a daily/weekly/fortnightly basis (depending on the patient's level of risk). This can later be sent to an *electronic patient record* repository provided by the GP (some of these policies are illustrated in Figure 7.7).

```
// handles alert event, e.g.~displays message on a plasma TV
on alert(msg) do
   /resources/alarm.display(msg)

// periodically updates electronic patient record
on time(00:00) do
    if updated == "true" then
        record = /resources/database.retrieveRecord();
                 /resources/surgery.updateRecord(record)
```

Figure 7.7: Example policies used in the data synthesis mission.

Data collection and synthesis mission is typically loaded by the patient SMC into his own home environment via a *hierarchical control style*. In cases where the patient is deemed to be in a more severe health condition, ECG monitoring may be prescribed by the doctor. In this case, the diabetes monitoring mission will contain an additional ECG monitoring mission, to be loaded into an ECG monitoring device available at the home environment (also via a *hierarchical control style*).

This scenario requires only relatively simple task-allocation styles, as typically *auction* of tasks or tasks loaded by *multiple managers* are not common in an e-health monitoring application. However, we identified the use of more complex patterns of task allocation in scenarios for collaborating teams of UAVs in search-and-rescue missions [SFLS$^+$08b].

7.5 Discussion

The case-study presented in this chapter described an application for the monitoring and treatment of diabetes. We demonstrated how the framework proposed in this thesis facilitates the construction of large policy-based SMC collaborations, as well as interactions across collaborations. We showed how policy-based collaborations are established between a number of SMCs representing sensors, diagnostic devices and other resources.

The use of styles promotes the systematic specification of these interactions. Architectural styles can capture standard solutions for composing SMCs, and allow others to reuse these solutions to resolve recurring problems encountered throughout the development of complex applications. The use of styles also enables the rigorous verification of these interactions and reasoning about their properties to guarantee the integrity of collaborating SMCs. However, the benefits of a software engineering approach are clearer if one considers the use of *application patterns* for specific domains, e.g. healthcare monitoring patterns. These rely on the general styles of structure, task-allocation and communication, but can be tailored and customised for a given scenario, defining application-specific functionality. Patterns can thus be reused and instantiated multiple times, using the sensors and resources available for each patient.

The formal model for SMC interactions which was specified in Alloy can be used to analyse the behaviour of SMCs in this scenario. The ability to analyse SMC interactions is crucial in order to verify whether they are correctly specified and deployed and whether they achieve the behaviour that is intended for them. Types of analysis that can be used include: *(a)* ensure a given SMC is bound to a particular abstraction, e.g. *sensor* and *insulin pump* SMCs should always be encapsulated and managed by a *patient* SMC; and *(b)* whenever an architectural style is deployed, all the roles bound to that style must be already assigned, ensuring the application has all the required devices to operate. The manner in which these properties can be verified using our formal model for SMC interactions was discussed in greater detail in Chapter 5.

Chapter 8

Discussion and Critical Evaluation

This chapter presents a more general discussion of the framework proposed in this thesis, in terms of its *usability*, *scalability* and *extensibility*. The limitations and deficiencies of the framework are also discussed.

8.1 Usability

Usability 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 styles satisfies these two issues when deploying SMC interactions. The parameterisation and instantiation of an individual architectural style typically requires about 10 lines or less of *PonderTalk* code. For comparison, the same interaction written manually without the aid of styles 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 architectural styles, by instantiating a single managed object which encapsulates the required support.

Manually setting up an interaction, for example an event sharing scheme similar to the *SharedBus* style, 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 abstractions for correctly setting up policies for event forwarding and installing the required *event templates* in disparate locations to handle these events. In contrast, the architectural style automatically defines the algorithm or protocol for the semantics for a given interaction, and distributes fragments of it to distributed SMCs according to their roles.

8.2 Scalability

The use of architectural styles reduces the complexity and size of the interactions, by structuring and decreasing the number of necessary bindings between SMCs. For example, the size of an interaction for event sharing between SMCs can be drastically reduced by using an abstraction such as the *SharedBus* style, when compared to a straightforward forwarding of events. Table 8.1 shows a comparison between the number of necessary bindings to achieve the event exchanges in both cases.

Table 8.1: Comparison between the number of bindings using a *SharedBus* style and a simple approach for event forwarding

#SMCs	*#Bindings (Diffusion)*	*#Bindings (SharedBus)*
2	1	1
3	3 = 2 + 1	2
4	6 = 3 + 2 + 1	3
5	10 = 4 + 3 + 2 + 1	4
6	15 = 5 + 4 + 3 + 2 + 1	5
7	21 = 6 + 5 + 4 + 3 + 2 + 1	6
...	...	...
n	(n - 1) + (n - 2) + (n - 3) + ... + (n - n)	n - 1

A similar 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 indicated in Figure 8.1 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 styles 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 architectural styles and patterns thus provides a measurable gain over unstructured solutions.

8.3 Extensibility

The categories of architectural styles can be seen as complementary perspectives for modelling the structural, task-allocation and communication aspects of an interaction. The catalogue presented in this thesis certainly is not complete. Instead we have focused solely on the frequently occurring styles that facilitate the design and composition of complex SMCs in our application scenarios.

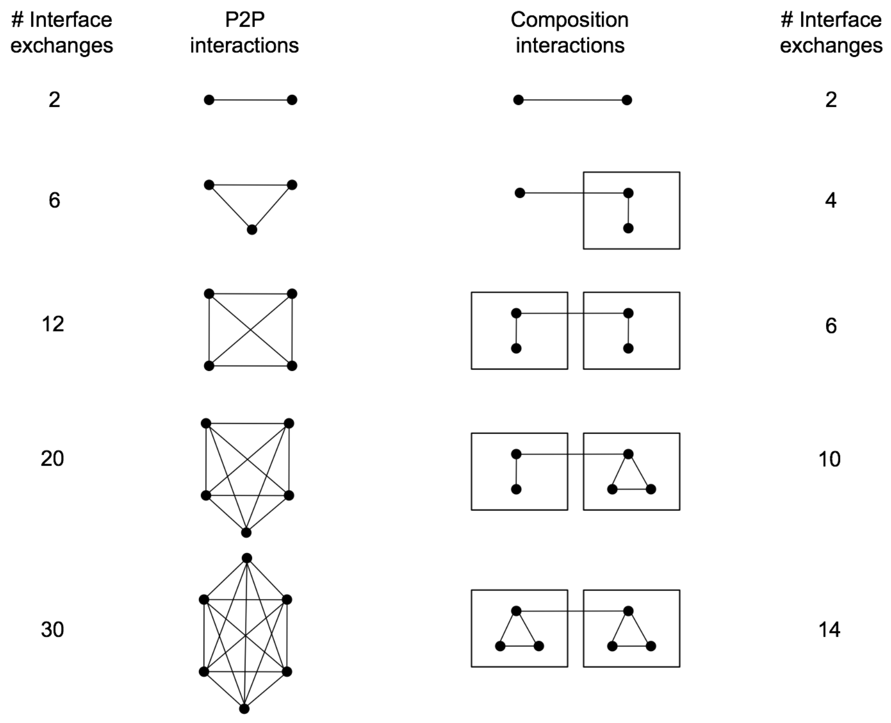


Figure 8.1: Interface exchanges in compositions and peer-to-peer interactions

The framework for SMC interactions can be extended in three different ways:

1. By recombining existing architectural styles into patterns of interactions, using a specific set of roles and architectural styles and their customisations, e.g. events to be forwarded, methods to be filtered and missions to be loaded. These complex interactions are specified solely in *PonderTalk*, and can be subsequently reused to form parts of a larger and more complex interaction.

2. By programming additional *managed objects* defining new architectural styles which can be added to the library of styles. These new managed objects must extend adequate classes in our framework and implement the abstract methods which they define. The definition of a new architectural style requires considerable programming effort, however, once implemented and checked that it conforms with the formal specification of SMC interactions, it can be reused as necessary.

3. By subclassing existing architectural styles to create more specialised

ones that can be included in the library for subsequent reuse. Typically, architectural styles define a generalised protocol or algorithm through which the exchange of events, interfaces and policies is achieved. Flexibility is obtained in the implementation because styles can be parameterised with *PonderTalk* blocks, which concretely define parts of the protocol or algorithm defined by the style. This was indicated by the correlation function which parameterises the *correlator* style, or the filtering function which parameterises the *composition* style, both discussed in Section 6.2.3. Based on this a specialised style *managed object* can predefine a concrete built-in function for *interface filtering*, *event correlation*, *mission bidding*, etc, which builds on top of the more general mechanism defined in the corresponding super class.

We expect that the investigation of other scenarios could result in the identification of additional abstractions for SMC interactions as many styles and patterns of interaction are specific to the application domain. Styles tailored to particular application scenarios, such as care management for a set of patients, can thus be defined and included in our library of styles.

8.4 Limitations

The work presented in this thesis is by no means complete, and a number of limitations and deficiencies were identified throughout its development. These are discussed in the following.

8.4.1 Security Mechanisms

Section 3.6 briefly discussed the minimum security requirements we identified for collaborations between SMCs and we illustrated the security management aspects of SMC interactions using these minimum requirements. However, the development of security management protocols for SMC interactions is outside the scope of this thesis. The work on *doctrines* [Keo05, KLS04] has previously investigated the formation of policy-based communities of autonomous nodes, focusing on the security protocols for realising trust-based authentication, access control and membership management. Our framework does not address privacy issues or cryptographic protection of exchanged data as work on these within the context of SMCs is being investigated elsewhere [KLS09, ZSLK09].

Instead, we focused on the definition of a framework for specification, instantiation and reuse of SMC interactions, which can accommodate additional security requirements as needed through the addition of new management roles which can then be used to address the security needs of a particular application, e.g. threshold cryptography [ZH99], intrusion detection [Sun96], DDoS detection [TSD07].

8.4.2 Dynamic Restructuring

This thesis focused on providing abstractions for systematically designing and establishing larger policy-based SMC interactions from simpler ones. The framework can accommodate dynamic discovery and departure of SMCs, dynamic reassignment of roles to the available SMCs, and dynamic loading and unloading of policies.

However, changes in the *requirements* of scenarios and applications may require a new configuration of architectural styles to replace an old one. This involves the re-specification and re-checking of an entire new interaction before it can replace an old interaction. Currently this is not done automatically, as interaction specification and translation from *PonderTalk* to *Alloy* notation for the purposes of model-checking is performed manually. Collaborating SMCs may also need to synchronise and achieve a *safe state* [KM90], e.g. all events forwarded were already delivered, before their current interaction configuration can be replaced. This still requires further investigation.

8.4.3 Model-Checking and PonderTalk to Alloy Translation

The declarative specification style used in *Alloy* was simple to use and the associated tool set offered rapid feedback on the model specification. The analyser is used both *(a)* to validate universal assertions with respect to general SMC interactions, e.g. a composed SMC never establishes interactions outside the composition, as well as *(b)* to check logical properties automatically for a specific instance of SMC interaction. The former requires an exhaustive search through a user-specified scope on the number of objects to be used, which makes the problem finite and thus reducible to a boolean formula, and depending on the size of the problem this kind of analysis may take several hours to complete. However, these are not expected to be performed online, as the purpose of this kind of analysis is to verify whether the implementation of the model itself correctly expresses the semantics of SMC interactions and whether unorthodox configurations of SMCs are ruled out of the model.

In contrast, the verification of logical properties for a specific instance of SMC collaboration is a much faster and more straightforward analysis. The predicates for the verification of properties receive as *input* the precise instance we want to verify, containing all the SMCs, interfaces, policies, etc, to evaluate the logical property. This requires expressing a specific interaction, e.g. all the SMCs available, their interfaces, the policies we want to load and the architectural styles that we want to use, in Alloy notation. Currently, the translation of a *PonderTalk* specification into *Alloy* notation to be used as input of these logical predicates has to be done manually. This is the main reason why we only use *Alloy* for verifying interactions prior to deployment in physical devices. However, if translation between *PonderTalk* into *Alloy* can be done automatically, dynamic changes in the SMC could be automatically translated and verified in *Alloy* during run-time. However, model checking of specifications still requires significant computational resources.

8.4.4 Prototype Implementation

The prototype has demonstrated the feasibility of the concepts presented in this thesis. However, this prototype can be improved in many ways with added tool support and a specification editor. The specification of SMC interactions and definition of patterns are written manually in *PonderTalk*, with no syntax validation or guidance in correctly specifying the interactions. Additionally, in the current implementation an interaction specification written in *PonderTalk* needs to be manually loaded into a physical device, e.g. SSH to a Gumstix and copy the specification file into it, which will then discover and bootstrap the interactions with other SMCs according to the specification file. We recognise that improved tool support could aid the specification of these interactions, as well as their automatic distribution and deployment into remote devices.

Our implementation runs on JamVM version 1.4.5 and GNU Classpath version 0.91, which are supported both in Gumstix and in Koala robots. However, JamVM does not run on BSN sensors which are much more resource-constrained, and we were not able to run our prototype on these devices. Although previous work has defined a lightweight implementation of the policy service that can run on BSNs [K^{+}07], our prototype assumes the existence of more powerful devices (similar to a Gumstix) capable of running the JamVM and thus the full-blown SMC framework.

Chapter 9

Conclusions

This chapter describes a summary of the work presented in this thesis, and discusses what has been achieved. The directions for future work resulting from the evaluation of the framework presented in the previous chapter are also outlined, and finally the concluding remarks are presented.

9.1 Review and Discussion of Achievements

Management in ubiquitous systems cannot rely on human intervention and centralised decision-making functions. This is due to their complexity and because these systems are composed of resources and devices which are inherently mobile and cannot refer to centralised management applications for their reconfiguration. The *Self-Managed Cell* provides an infrastructure for the management of autonomous components which is based on a policy-driven feedback control-loop, where policies provide the means for adapting the functioning of an SMC in response to changes in its own context, e.g. failures of components or performance degradation. Individual SMCs thus form devolved administrative domains charged with enforcing local decision rules that govern their own behaviour. However, applications typically consist of ad-hoc collaborations of devices and resources and therefore require elementary SMCs to negotiate the necessary interactions with other components in their surroundings and be assembled into larger and more complex structures based on the same principles of self-management. Most of the examples used in this thesis are derived from requirements in the context of healthcare monitoring applications comprising autonomous sensors and actuators and in the self-management of teams of unmanned autonomous vehicles. However, the principles and results are more generally applicable to other ubiquitous scenarios, such as in the management of large virtual organisations.

This thesis presented an integrated framework which supports the design and the rapid establishment of policy-based interactions by systematically composing simpler abstractions as building elements of a more complex collaboration, using the Self-Managed Cell as the underlying infrastructure. 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.

We first identified the underlying principles for supporting SMC interactions. Through careful analysis of our application scenarios we observed that these interactions rely on essentially the same three basic exchange mechanisms: *(a)* the exchange of *policies*; *(b)* the exchange of *events* required for triggering policies; and *(c)* the exchange of *interfaces* which are required for validating the actions prescribed by policies. These laid the groundwork for supporting the composition and federation of SMCs. In particular, the notion of missions is used to define a group of policies which are loaded into a remote SMC in order to prescribe how it should behave within the context of an interaction. A key observation is that a mission is akin to a form of constrained programming, and thus the framework caters for the careful verification of a mission before its policies are granted permission to execute on a remote SMC. These observations have been previously described in [SFLD+07, SFL07].

Basic SMC interactions can be used for the management of the application-specific aspects of an application, e.g. healthcare monitoring, reconnaissance and rescue operations, as well as for the management aspects of how SMC interactions themselves are realised. We illustrated an example of how SMCs can be used for realising the security management of their own interactions and this revealed itself to be a key decision. As a result, we identified a number of management tasks which are needed for supporting secure SMC interactions, but more importantly, we demonstrated that the infrastructure is flexible and can accommodate additional security functions according to the requirements of a particular collaboration. This was presented in detail in [SFLS$^+$08a].

Building elaborate applications using solely these elementary abstractions can be difficult to manage and deploy and we advocate the use of software engineering principles, such as patterns and software architectures, for systematically building policy-based SMC interactions. We defined a catalogue of reusable patterns and provided a more general infrastructure for specifying, instantiating and reusing SMC interactions. Patterns support the definition of specific algorithms and protocols which govern how the exchanges of policies, events and interfaces must be achieved, and are used to define independently the task-allocation, communication and structural aspects of an interaction. Patterns can then be composed in a methodical manner thereby facilitating the engineering of larger policy-based SMC interactions. The use of patterns was first proposed in [SFLS09].

We implemented a framework which realises SMC interactions and includes both the specification, establishment and operation of simple SMC interactions, as well as a library of reusable architectural styles for composing and federating SMCs. Our evaluation demonstrates that the policy-based infrastructure scales down and can be used effectively in constrained devices with limited computational power and memory. This was confirmed by a practical evaluation of the performance of our framework executing in these devices. Our evaluation also indicates that the use of patterns for engineering SMC interactions mitigates the problems of scaling to larger systems by reducing both the programming complexity and the complexity of the interactions established between components during run-time. A case-study scenario presenting how SMCs were used for building a policy-based application for diabetes monitoring and treatment was presented in order to complement the functional evaluation of our work, demonstrating how real-world applications can be realised using our framework.

The successful operation of SMC collaborations, however, depends both on their correct specification and on the suitability of the participating SMCs. We modelled in Alloy a formal specification of the SMC behaviour, which makes possible the rigorous verification of specific SMC interactions. Interactions can be checked automatically to guarantee that SMCs are able to enforce their policies before these interactions are deployed in physical devices, e.g. smartphones, Gumstix, sensors. Part of this formal specification was presented in [SFLSE09].

9.2 Future Work

Several limitations have been identified in the discussion presented in the previous chapter. We now consider directions for future work.

The security aspects for establishing SMC interactions must be adequately addressed. This thesis briefly discussed the minimum security requirements, but further work needs to investigate the development of appropriate *security management protocols* for SMC interactions. Work on this direction has started to be addressed elsewhere [KLS09, ZSLK09, Keo05, KLS04].

Our overall goal is to be able to form compositions and federations of SMCs dynamically, suitable to particular application scenarios, such as care management for a set of patients, by instantiating combinations of pre-defined patterns. A promising research direction is the identification of *application patterns* for specific domains, e.g. *monitoring patterns*, or *data analysis patterns*. These will rely on the basic patterns of structure, task-allocation and communication, and add to them application-specific functionality. A mission, which has been hitherto used as a means of packaging a set of policies, could have its behaviour augmented by supporting the specification and enforcement of application patterns that are transferred into an SMC.

The framework presented in this thesis provides the mechanisms for specifying and instantiating SMC interactions, however applications still have to be "built" manually from styles, missions, interfaces, etc. Support for the automatic generation of SMC interactions and the use of goal refinement and planning techniques is still a focus of active research. These could be used to select automatically the abstractions in a specific application scenario. High-level user requirements, for example *ECG monitoring*, could then be refined into concrete federations of SMCs using the required policies and the mechanisms for realising the corresponding application.

9.3 Closing Remarks

Devolved management is the key for addressing the complexity of large-scale ubiquitous systems which are formed as collaborations of smaller, yet autonomous, components. The work presented in this thesis investigated how policy-based autonomous components can be federated and composed to form larger ubiquitous applications. This thesis relied on previous research to address a new problem: *engineering policy-based ubiquitous 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. This is built upon previous work on policy-based management developed at Imperial College over the past 20 years.

The use of patterns for systematically building policy-based systems is a novel and promising approach. Although we have concentrated on the *Self-Managed Cell* framework, the principles and techniques proposed in this thesis can benefit the engineering process of pervasive and autonomous systems in general.

Bibliography

[A$^+$02] Iara Augustin et al. ISAM - a software architecture for adaptive and distributed mobile applications. In *Proceedings of the 7th IEEE Symposium on Computers and Communications*, pages 333 – 338, Taormina, Italy, July 2002. IEEE Computer Society.

[ACG$^+$05] Dakshi Agrawal, Seraphin Calo, James Giles, Kang-Won Lee, and Dinesh Verma. Policy management for networked systems and applications. In *Proceedings of the 9th IFIP IEEE International Symposium on Integrated Network Management*, pages 455 – 468, Nice, France, May 2005. IEEE Computer Society.

[ADK$^+$06] Eskindir Asmare, Naranker Dulay, Hansang Kim, Emil Lupu, and Morris Sloman. Management architecture and mission specification for unmanned autonomous vehicles. In *Proceedings of the 1st SEAS DTC Technical Conference*, Edinburgh, Scotland, 2006.

[ADL$^+$07] Eskindir Asmare, Naranker Dulay, Emil Lupu, Morris Sloman, Seraphin Calo, and Jorge Lobo. Secure dynamic community establishment in coalitions. In *Proceedings of the Military Communications Conference (MILCOM)*, pages 1–7, Orlando, FL, 2007.

[AG94] Robert Allen and David Garlan. Beyond definition/use: architectural interconnection. In *Proceedings of the workshop on Interface definition languages*, pages 35–45, New York, NY, USA, 1994. ACM.

[AGLL05] Dakshi Agrawal, James Giles, Kang-Won Lee, and Jorge Lobo. Policy ratification. In *Proceedings of the Sixth IEEE International*

Workshop on Policies for Distributed Systems and Networks (Policy), pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[AHW04] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS)*, pages 8–12, New York, NY, USA, 2004. ACM Press.

[AL98] Yariv Aridor and Danny B. Lange. Agent design patterns: elements of agent application design. In *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, pages 108–115, New York, NY, USA, 1998. ACM.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[AMCRC04] Jalal Al-Muhtadi, Shiva Chetan, Anand Ranganathan, and Roy Campbell. Super spaces: A middleware for large-scale pervasive computing environments. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops (PERCOMW)*, page 198, Washington, DC, USA, 2004. IEEE Computer Society.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743/2003 of *Lecture Notes in Computer Science*, pages 179–204, Darmstadt, Germany, July 2003. Springer Berlin / Heidelberg.

[Ban05] Arosha K Bandara. *A Formal Approach to Analysis and Refinement of Policies*. PhD thesis, Imperial College London, London, U.K., July 2005.

[BBC$^+$03] David Bantz, Chatschik Bisdikian, David Challener, John Karidis, Steve Mastrianni, Ajay Mohindra, Dennis Shea, and Michael

Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003.

[BC04] Gregory Biegel and Vinny Cahill. A framework for developing mobile, context-aware applications. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 361, Washington, DC, USA, 2004. IEEE Computer Society.

[BLR03] Arosha Bandara, Emil Lupu, and Alessandra Russo. Using event calculus to formalise policy sepcification and analysis. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy)*, Como, Italy, June 2003. IEEE.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[CFP+06] Marinos Charalambides, Paris Flegkas, George Pavlou, Javier Rubio-Loyola, Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Morris Sloman, and Naranker Dulay. Dynamic policy analysis and conflict resolution for diffserv quality of service management. In Joseph L. Hellerstein and Burkhard Stiller, editors, *NOMS*, pages 294–304. IEEE, 2006.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structure (FoSSaCS)*, pages 140–155, London, UK, 1998. Springer-Verlag.

[CH97] Alper Caglayan and Colin Harrison. *Agent sourcebook*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[CHL+04] Eric Chung, Jason Hong, James Lin, Madhu Prabaker, James Landay, and Alan Liu. Development and evaluation of emerging design patterns for ubiquitous computing. In *Proceedings of the 5th Conference on Designing Interactive Systems (DIS)*, pages 233–242, New York, NY, USA, 2004. ACM.

[CKC04] Luiz Chaimowicz, Vijay Kumar, and Mario F. M. Campos. A paradigm for dynamic coordination of multiple robots. *Autonomous Robots*, 17(1):7–21, July 2004.

[Cle96] Paul Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD)*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[CLM$^+$09] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, Arosha Bandara, Seraphin Calo, and Morris Sloman. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Sydney, Australia, March 2009. IEEE.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, 1987.

[D$^+$01] Nicodemos Damianou et al. The Ponder policy specification language. In *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (Policy)*, pages 18–39, Bristol, U.K., January 2001. IEEE Computer Society.

[DHL$^+$05] Naranker Dulay, Steven Heeps, Emil Lupu, R Mathur, O Sharma, Morris Sloman, and Joe Sventek. AMUSE: autonomic management of ubiquitous e-health systems. In *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK, September 2005.

[DJS08] Steven Davy, Brendan Jennings, and John Strassner. Efficient policy conflict analysis for autonomic network management. In *Proceedings of the Fifth IEEE Workshop on Engineering of Autonomic* *and Autonomous Systems (EASE)*, pages 16–24, Washington, DC, USA, 2008. IEEE Computer Society.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[DLS+05] Naranker Dulay, Emil Lupu, Morris Sloman, Joe Sventek, Nagwa Badr, and Steven Heeps. Self-managed cells for ubiquitous systems. In *Proceedings of the 3rd International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security*, volume 3685 of *LNCS*, pages 1–6, St. Petersburg, Russia, September 2005.

[DLSD01] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicodemos Damianou. A policy deployment model for the Ponder language. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, pages 529–543, Seattle, Washington, USA, May 2001. IEEE.

[DOKvO99] Dwight Deugo, Franz Oppacher, J. Kuester, and Ingo von Otte. Patterns as a means for intelligent software engineering. In *Proceedings of the International Conference on Artificial Intelligence (IC-AI)*, volume 2, pages 605–611, 1999.

[Don02] Jing Dong. UML extensions for design pattern compositions. *Journal of Object Technology*, 1(5):151–163, November-December 2002.

[Don03] Jing Dong. Representing the applications and compositions of design patterns in UML. In *Proceedings of the 2003 ACM symposium on Applied computing (SAC)*, pages 1092–1098, New York, NY, USA, 2003. ACM.

[Don04] Jing Dong. Adding pattern related information in structural and behavioral diagrams. *International Journal of Information and Software Technology (IST*, 2004:293–300, 2004.

[EHRLR80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and Dabbala Rajagopal Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253, 1980.

[Fal03] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 27–34, New York, NY, USA, 2003. ACM.

[FDC01] Adrian Friday, Nigel Davies, and Elaine Catterall. Supporting service discovery, querying and interaction in ubiquitous computing environments. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 7–13, Santa Barbara, U.S., 2001.

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 196–205. ACM, 2005.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[GBCC02] Adrian Fitzpatrick Gregory, Gregory Biegel, Siobhn Clarke, and Vinny Cahill. Towards a sentient object model. In *In Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE'2002*, 2002.

[GC03] Alan Ganek and Thomas Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[GCH$^+$04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[GCK02] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44(1):23–49, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1st edition, 1995. 416 pages.

[Gri02] Robert Grimm. *System support for pervasive applications*. PhD thesis, University of Washington, Washington, USA, Dec. 2002.

[Gri04] Robert Grimm. One.world: experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, Jul.-Sep. 2004.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[GSSS02] David Garlan, Daniel Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, Apr.-Jun. 2002.

[HBM98] Richard Hayton, Jean Bacon, and Ken Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, pages 3 – 14, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[HCP07] Antonis M. Hadjiantonis, Marinos Charalambides, and George Pavlou. A policy-based approach for managing ubiquitous networks in urban spaces. In *Proceedings of the IEEE International*

Conference on Communications (ICC), pages 2089–2096, Glasgow, Scotland, 2007. IEEE.

[HCY99] Sandra Hayden, Christina Carrick, and Qiang Yang. Architectural design patterns for multi-agent coordination. In *Proceedings of the International Conference on Agent Systems (Agents)*, Seattle, WA, May 1999.

[HKR08] Vincent Hilaire, Abder Koukam, and Sebastian Rodriguez. An adaptative agent architecture for holonic multi-agent systems. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1–24, 2008.

[HMP06] Antonis M. Hadjiantonis, Apostolos Malatras, and George Pavlou. A context-aware, policy-based framework for the management of MANETs. In *Proceedings of the 7th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy)*, pages 23–34, Washington, DC, USA, 2006. IEEE Computer Society.

[Hor01] Paul Horn. Autonomic computing: IBM perspective on the state of information technology. Presented at AGENDA 2001, Scottsdale, AR, October 2001. Available at: http://www.research.ibm.com/autonomic/.

[HP03] HP. HP utility data center: Enabling enhanced datacenter agility. http://www.hp.com/large/globalsolutions/ae/pdfs/udc enabling.pdf, May 2003.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*, chapter Predicate Logic. Cambridge University Press, 2004.

[HYBM00] John A. Hine, Walt Yao, Jean Bacon, and Ken Moody. An architecture for distributed OASIS services. In *Proceedings of the Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 104–120, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[INPS03] Luca Iocchi, Daniele Nardi, Maurizio Piaggio, and Antonio Sgorbissa. Distributed coordination in heterogeneous multi-robot systems. *Autonomous Robots*, 15(2):155–168, September 2003.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[JFW02] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67 – 74, April-June 2002.

[JLT$^+$08] Somesh Jha, Ninghui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[K$^+$07] Sye-Loong Keoh et al. Policy-based management for body-sensor networks. In *Proceedings of the 4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN)*, LNCS, pages 92 – 98, Aachen, Germany, Mar. 2007. Springer.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan 2003.

[KDL$^+$07] Sye-Loong Keoh, Naranker Dulay, Emil Lupu, Kevin Twidle, Alberto E. Schaeffer-Filho, Morris Sloman, Steven Heeps, Stephen Strowes, and Joe Sventek. Self-managed cell: A middleware for managing body-sensor networks. In *Proceedings of the 4th Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous)*, pages 1–5, Washington, DC, USA, 2007. IEEE Computer Society.

[Keo05] Sye-Loong Keoh. *A Policy-based Security Framework for Ad-hoc Networks*. PhD thesis, Imperial College London, London, UK, 2005.

[KFJ03] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy)*, page 63, Washington, DC, USA, 2003. IEEE Computer Society.

[KGM02] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII (ATAL)*, pages 128–140, London, UK, 2002. Springer-Verlag.

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KKPS98] Elizabeth A. Kendall, P. V. Murali Krishna, Chirag V. Pathak, and C. B. Suresh. Patterns of intelligent and mobile agents. In *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, pages 92–99, New York, NY, USA, 1998. ACM.

[KLS04] Sye-Loong Keoh, Emil Lupu, and Morris Sloman. PEACE: A policy-based establishment of ad-hoc communities. In *Proceedings of the*

20th Annual Computer Security Applications Conference (ACSAC), pages 386–395, Washington, DC, 2004. IEEE Computer Society.

[KLS09] Sye-Loong Keoh, Emil Lupu, and Morris Sloman. Securing body sensor networks: Sensor association and key management. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, pages 1–6, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. *fose*, 0:259–268, 2007.

[Kra90] Jeff Kramer. Configuration programming - a framework for the development of distributable systems. In *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering (CompEuro)*, pages 374 – 384. IEEE, May 1990.

[KZ03] Tim Kindberg and Kan Zhang. Secure spontaneous device association. In *Proceedings of the 5th International Conference of Ubiquitous Computing (UbiComp)*, Seattle, WA, USA, 2003.

[LB03] James A. Landay and Gaetano Borriello. Design patterns for ubiquitous computing. *Computer*, 36(8):93–95, 2003.

[LBN99] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *Proceedings of the 16th National Conference on Artificial Intelligence*, pages 291 – 298, Orlando, FL, July 1999.

[LDS$^+$08] Emil Lupu, Naranker Dulay, Morris Sloman, Joe Sventek, Steven Heeps, Stephen Strowes, Kevin Twidle, Sye-Loong Keoh, and Alberto Schaeffer-Filho. AMUSE: autonomic management of ubiquitous systems for e-health. *Concurrency and Computation: Practice and Experience, John Wiley*, 20(3):277–295, May 2008.

[Lin01] Peter Linington. Distributed systems, an ODP perspective. *Formal methods for distributed processing: a survey of object-oriented approaches*, pages 18–35, 2001.

[LKA$^+$95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[LMR98] Peter Linington, Zoran Milosevic, and Kerry Raymond. Policies in communities: Extending the ODP enterprise viewpoint. In *In proceedings of the 2nd International Workshop on Enterprise Distributed Object Computing (EDOC)*, pages 14–24, November 1998.

[LS99] Emil Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, Nov. - Dec. 1999.

[LSDD00] Emil Lupu, Morris Sloman, Naranker Dulay, and Nicodemos Damianou. Ponder: realising enterprise viewpoint concepts. In *Proceedings of the 4th International Conference on Enterprise Distributed Object Computing*, pages 66–75, Makuhari, Japan, Sept 2000. IEEE.

[Lun93] Teresa F. Lunt. A survey of intrusion detection techniques. *Computers and Security*, 12(4):405–418, 1993.

[Lup98] Emil Lupu. *A Role-Based Framework for Distributed Systems Management*. PhD thesis, Imperial College London, July 1998.

[LWF01] Antónia Lopes, Michel Wermelinger, and José Luiz Fiadeiro. A compositional approach to connector construction. In *WADT '01: Selected papers from the 15th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 201–220, London, UK, 2001. Springer-Verlag.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the*

5th European Software Engineering Conference, pages 137–153, London, UK, 1995. Springer-Verlag.

[MESW01] Bob Moore, Ed Ellesson, John Strassner, and Andrea Westerinen. Policy core information model, version 1 specification. request for comments 3060, network working group. Available at: http://www.ietf.org/rfc/rfc3060.txt, 2001.

[MKMG97] Robert Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, January 1997.

[MMP00] Nikunj Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[NL98] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *Lecture Notes In Computer Science*, 1419:61–91, 1998.

[Oak01] Scott Oaks. *Java Security*, chapter 1-3, pages 1–57. O'Reilly, 2nd edition, 2001.

[OGT$^+$99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[Oxy06] Oxygen website, 2006. Available at: <http://www.oxygen.lcs.mit.edu/>.

[PAVB02] Maurice Puyau, Anne Adolph, Firoz Vohra, and Nancy Butte. Validation and calibration of physical activity monitors in children. *Obesity Research*, 10(3):150–157, 2002.

[Phi06] Andrew Phillips. *Specifying and Implementing Secure Mobile Applications in the Channel Ambient System*. PhD thesis, Imperial College London, April 2006.

[PJKF03] Shankar Ponnekanti, Brad Johanson, Emre Kiciman, and Armando Fox. Portability, extensibility and robustness in iROS. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, pages 11 – 19, Dallas-Fort Worth Metroplex, Texas, March 2003.

[PW92] Dewayne Perry and Alexander Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RDD07] Giovani Russello, Changyu Dong, and Naranker Dulay. Authorisation and conflict resolution for hierarchical domains. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks (Policy)*, Bologna, Italy, Jun. 2007. IEEE CS-Press.

[RG95] Anand Rao and Michael Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, 1995.

[RHC$^+$02] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[RHK03] Sebastian Rodriguez, Vincent Hilaire, and Abderrafiaa Koukam. Towards a methodological framework for holonic multi-agent systems. In *Proceedings of the ESAW 03 workshop*, pages 179–185, 2003.

[RLSC+05] Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, George Pavlou, and Alberto Lluch-Lafuente. Using linear temporal model checking for goal-oriented policy refinement frameworks. In *IEEE International Workshop on Policies for Distributed Systems and Networks (Policy)*, pages 181–190. IEEE Computer Society, 2005.

[RP03] Gruia-Catalin Roman and Jamie Payton. Mobile UNITY schemas for agent coordination. In *Proceedings of the 10th International Workshop on Abstract State Machines*, volume 2589 of *LNCS*, pages 126–150. Springer, March 2003.

[S+06] Stephen Strowes et al. An event service supporting autonomic management of ubiquitous systems for e-health. In *Proceedings of the 5th International Workshop on Distributed Event-Based Systems*, Lisbon, Portugal, July 2006.

[SA00] Frank Stajano and Ross J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pages 172–194, London, UK, 2000. Springer-Verlag.

[SAL06] John Strassner, Nazim Agoulmine, and Elyes Lehtihet. FOCALE - a novel autonomic networking architecture. In *Latin American Autonomic Computing Symposium*, Campo Grande, MS, Brazil, July 2006.

[San96] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*, page 9, New York, NY, USA, 1996. ACM Press.

[Sat01] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug. 2001.

[SB98] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering Methodologies*, 11(2):215–255, 2002.

[SB08] Jatinder Singh and Jean Bacon. Event-based data control in healthcare. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 84–86, New York, NY, USA, 2008. ACM.

[SBD$^+$05] Joseph Sventek, Nagwa Badr, Naranker Dulay, Steven Heeps, Emil Lupu, and Morris Sloman. Self-managed cells and their federation. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering*, pages 97–107, Porto, Portugal, June 2005.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[SBS$^+$02] Dirk Balfanz Smetters, Dirk Balfanz, D. K. Smetters, Paul Stewart, and H. Chi Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, San Diego, California, 2002.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[SDK$^+$95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 2–10, Annapolis, MD, USA, 1996. IEEE Computer Society.

[SFL07] Alberto Schaeffer-Filho and Emil Lupu. Abstractions to support interactions between self-managed cells. In *Proceedings of the 1st International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, LNCS, pages 160–163, Oslo, Norway, June 2007. Springer.

[SFLD$^+$07] Alberto Schaeffer-Filho, Emil Lupu, Naranker Dulay, Sye-Loong Keoh, Kevin Twidle, Morris Sloman, Steven Heeps, Stephen Strowes, and Joe Sventek. Towards supporting interactions between self-managed cells. In *Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 224–233, Boston, USA, July 2007. IEEE Computer Society.

[SFLS$^+$08a] Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, Sye-Loong Keoh, Jorge Lobo, and Seraphin Calo. A role-based infrastructure for the management of dynamic communities. In *Proceedings of the 2nd International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, LNCS, pages 1–14, Bremen, Germany, July 2008. Springer.

[SFLS$^+$08b] Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, Jorge Lobo, and Seraphin Calo. Engineering abstractions for modelling interactions between self-managed cells (short paper). In *Proceedings of the 2nd Annual Conference of ITA (ACITA)*, September 2008.

[SFLS09] Alberto Schaeffer-Filho, Emil Lupu, and Morris Sloman. Realising management and composition of self-managed cells in pervasive healthcare. In *Proceedings of the 3rd International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth)*, London, UK, April 2009. IEEE.

[SFLSE09] Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, and Susan Eisenbach. Verification of policy-based self-managed cell interactions using Alloy. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (Policy)*, London, UK, July 2009. IEEE.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[SL02] Morris Sloman and Emil Lupu. Security and management policy specification. *IEEE Network*, 16(2):10–19, Mar.-Apr. 2002.

[Slo94a] Morris Sloman. *Netword and Distributed Systems Management*, chapter Specification of Management Policies and Discretionary Access Control, pages 455 – 480. Addison-Wesley, 1994.

[Slo94b] Morris Sloman. Policy-driven management for distributed systems. *Journal of Network and Systems Management*, 4(2):333–360, 1994.

[Smi88] Reid G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *Distributed Artificial Intelligence*, pages 357–366, 1988.

[SPP$^+$03] Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A case for goal-oriented programming semantics. In *System Support for Ubiquitous Computing (UbiSys)*, Seattle, Washington, October 2003.

[SS03] Asim Smailagic and Daniel Siewiorek. Application design for wearable and context-aware computers. *IEEE Pervasive Computing*, 1(4):20–29, Oct-Dec 2003.

[Sun96] Aurobindo Sundaram. An introduction to intrusion detection. *Crossroads*, 2(4):3–7, 1996.

[SVBM08] Jatinder Singh, Luis Vargas, Jean Bacon, and Ken Moody. Policy-based information sharing in publish/subscribe middleware. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks (Policy)*, pages 137–144, Washington, DC, USA, 2008. IEEE Computer Society.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[TMD09] Richard N. Taylor, Nenad Medvidovi, and Irvine E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, February 2009.

[TOH99] Yasuyuki Tahara, Akihiko Ohsuga, and Shinichi Honiden. Agent system development method based on agent patterns. In *ISADS '99: Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems*, page 261, Washington, DC, USA, 1999. IEEE Computer Society.

[TS88] Kevin Twidle and Morris Sloman. Domain-based configuration and name management for distributed systems. In *Proceedings of the Future Trends of Distributed Computing Systems in the 1990s*, pages 147–153, Hong Kong, Sep. 1988. IEEE.

[TSD07] Vrizlynn L. L. Thing, Morris Sloman, and Naranker Dulay. Non-intrusive IP traceback for DDoS attacks. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 371–373, New York, NY, USA, 2007. ACM.

[Tut09] Tutorial for Alloy Analyzer 4.0, 2009. Available at: <http://alloy.mit.edu/alloy4/tutorial4/>. Access in: February 2009.

[UBJ+03] Andrzej Uszok, Jeffrey Bradshaw, Renia Jeffers, Niranjan Suri, Patrick Hayes, Maggie Breedy, Larry Bunch, Matthew Johnson, Shriniwas Kulkarni, and James Lott. KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.

[UG02] Mihaela Ulieru and Adam Geras. Emergent holarchies for e-health applications: a case in glaucoma diagnosis. *IECON 02 [Industrial Electronics Society, IEEE 2002 28th Annual Conference of the]*, 4:2957–2961 vol.4, 5-8 Nov. 2002.

[UK09] Diabetes UK. Understanding diabetes: Your essential guide, 2009. [Online; accessed 22-April-2009].

[VCC+02] P. Veríssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. Cortex: Towards supporting autonomous and cooperating sentient entities. In *Proceedings of European Wireless 2002*, pages 595–601, Florence, Italy, feb 2002.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[Woo02] Michael Wooldridge. *Introduction to Multi-Agent Systems*. John Wiley & Sons, June 2002.

[YMB01] Walt Yao, Ken Moody, and Jean Bacon. A model of OASIS role-based access control and its support for active security. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access con-*

trol models and technologies, pages 171–181, New York, NY, USA, 2001. ACM.

[ZH99] Lidong Zhou and Zygmunt Haas. Securing ad hoc networks. Technical report, Cornell University, Ithaca, NY, USA, 1999.

[ZJW01] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Organizational abstractions for the analysis and design of multi-agent system. In *Proceedings of the 1st International workshop on Agent-oriented software engineering (AOSE)*, pages 235–251, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering Methodologies*, 12(3):317–370, 2003.

[ZSLK09] Yanmin Zhu, Morris Sloman, Emil Lupu, and Sye-Loong Keoh. Vesta: A secure and autonomic system for pervasive healthcare. In *Proceedings of the 3rd International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth)*, London, UK, 2009. IEEE.