Role Oriented Adaptive Design

Alan Colman

Submitted in fulfilment of the requirements
of the degree of Doctor of Philosophy

Faculty of Information and Communication Technologies
Swinburne University of Technology

October 2006

Abstract

Software systems are becoming inexorably more open, distributed, pervasive, mobile and connected. This thesis addresses the problem of how to build adaptive software systems. These systems need to reliably achieve system-level goals in volatile environments, where the system itself may be built from components of uncertain behaviour, and where the requirements for the software system may be changing. This thesis adopts the system-theoretic concept of *ontogenic adaptation* from biology, and applies it to software architecture. Ontogenic adaptation is the ability of an individual system to maintain its *organisational* integrity by reconfiguring and regulating itself. A number of approaches to adaptive software architecture have been recently proposed that, to varying degrees, enable limited adaptive behaviour and reconfiguration, but none possess all the properties needed for ontogenic adaptation. We introduce a meta-model and framework called Role Oriented Adaptive Design (ROAD) that is consistent with the concept of maintaining organisational integrity through ontogenic adaptation.

The ROAD meta-model defines software applications as networks of functional roles which are executed by players (objects, components, services, agents, people, or role-composites). These flexible organisational structures are adaptive because the relationships (contracts) between roles, and the bindings between roles and players, can be regulated and reconfigured at run-time. Such flexible organisational role-structures are encapsulated into composites each with its own organiser. Because self-managed composites are themselves role-players, these composites can be distributed and recursively composed. The organisers of the composites form a management system over which requirements and performance data pass. Rather than being monolithic constructions, ROAD software applications are dynamic, self-managed compositions of loosely-coupled, and potentially, distributed entities.

The concepts in the ROAD meta-model have been implemented in a programming framework which can be extended by the application programmer to create adaptive applications. Central to this framework are dynamic contracts. These contracts define the role structure, control interactions between the role instances, and measure the performance of those interactions. Adaptivity is achieved by monitoring and manipulating these contracts, along with the role-player bindings. Contracts have been implemented using the mechanism of “association aspects”.

The applicability of the ROAD framework to the domain of Service-Oriented Computing is demonstrated. The framework is further evaluated in terms of its ability to express the concept of ontogenic adaptation and also in terms of the overhead its runtime infrastructure imposes on interactions.

Declaration

Alan Colman
October 2006
Melbourne

Acknowledgements

My sincere thanks go to my supervisor Professor Jun Han, who agreed to take on a foundling PhD candidate who had some strange ideas on applying systems theory to software architecture. Thank you Jun for the many hours spent discussing the topic, for your guidance and for your assiduous reviewing of papers and drafts. My warm thanks also to my fellow research student, Linh Duy Pham. Many of the detailed decisions on the ROAD framework implementation were a collaborative result of discussions between Linh and myself. Linh also needs to be credited with taking over much of the coding of the ROAD framework, and for writing the test code.

My thanks to my initial supervisor, Dr Lorraine Johnston, who agreed to support my candidature and, prior to her retirement, gave me the freedom to explore domains of research that weren't necessarily her own. My thanks also to Professor Ryszard Kowalczyk for agreeing to act as associate supervisor.

Thanks to the members of the Component Software and Enterprise Systems research group at Swinburne, in particular Dr. Yan Jin, Antony Tang and Dr Jean-Guy Schneider for their feedback on my presentations. I also thank Assoc. Professor Phillipe Collet for his comments and clarifications on the ConFract framework. Acknowledgement is also due to the many anonymous reviewers of my conference and journal papers on which this thesis was based, and whose comments helped hone this work.

This project would not have been possible without the scholarships provided by the Australian and Victorian governments, and Swinburne University of Technology. I would also like to acknowledge the contribution of Microsoft Research (Europe) and the American Association of Artificial Intelligence in providing travel scholarships that enabled me to attend conferences.

Finally, I would like to express my profound gratitude to my wife Katherine Westfold, without whom this project would never have started. Thank you, Kathy, for your support; for the many hours pent sproofing rafts; and for finally convincing me to use semi-colons.

Contents

List of Figures

List of Tables

List of Publications

This thesis is largely based on the following peer-reviewed articles for which I am the primary author.

Journal Articles:

Colman, A. and Han, J. (to appear) 'Using Role-based Coordination to Achieve Software Adaptability', Science of Computer Programming, Elsevier.

Colman, A. and Han, J. (2006) 'Adaptive service-oriented systems: an organisational approach', International Journal of Computer Systems Science & Engineering, vol. 21, no. 4, CRL, pp.235-246.

Colman, A. and Han, J. (2006), "Using Associations Aspects to Implement Organisational Contracts," *Electronic Notes on Theoretical Computer Science,* vol. 150, no. 3, Elsevier, pp37-53.

Conference and Workshop Papers:

Colman, A., Pham, L.D., Han, J., and Schneider, J.-G. (forthcoming 2006) "Adaptive Application-Specific Middleware," In *Proceedings of the Middleware for Service Oriented Computing Workshop (MW4SOC 2006)* Melbourne, Australia, ACM.

Colman, A. and Han, J., (2005) "An organisational approach to building adaptive service-oriented systems," *Proceedings of First International Workshop on Engineering Service Compositions,WESC'05, in IBM Research Report RC23821,* Amsterdam, The Netherlands. (An extended version can be found in the CSSE paper listed above)

Colman, A. and Han, J., (2005) "Organizational Roles and Players," Proceedings of the *AAAI Fall Symposium, Roles, an interdisciplinary perspective,* Arlington, Virginia. (Selected for extension and potential inclusion in the journal: *Applied Ontology*)

Colman, A. and Han, J., (2005) "Using Associations Aspects to Implement Organisational Contracts," *Proceedings of the 1st International Workshop on Coordination and Organisation (CoOrg 2005),* Namur, Belgium. (An extended version can be found in the ENTCS paper listed above)

Colman, A. and Han, J., (2005) "Coordination Systems in Role-based Adaptive Software," *Proceedings of the 7th International Conference on Coordination Models and Languages (COORD 2005), LNCS 3454,* Namur, Belgium. (Selected for extension and inclusion in the journal: *Science of Computer Programming*)

Colman, A. and Han, J., (2005) "On the autonomy of software entities and modes of organisation," *Proceedings of the 1st International Workshop on Coordination and Organisation (CoOrg 2005)* , Namur, Belgium.

Colman, A. and Han, J., (2005) "Operational management contracts for adaptive software organisation," *Proceedings of the Australian Software Engineering Conference (ASWEC 2005),* Brisbane, Australia.

Colman, A. and Han, J., (2005) "Organizational abstractions for adaptive systems," *Proceedings of the 38th Hawaii International Conference of System Sciences,* Hawaii, USA.

Part I

Adaptation and Organisation

1

Introduction

Software systems are becoming inexorably more open, distributed, pervasive, mobile and connected. This accelerating trend is being driven by new technologies that take advantage of the rapidly increasing power and falling cost of hardware and networking. Such systems have to interact with other systems and within environments that are becoming increasingly heterogeneous and dynamic. This thesis addresses the problem of how to build adaptive software systems that can reliably achieve system-level goals in environments that are volatile, where the system itself may be built from components of uncertain behaviour, and where the requirements for the software system may be changing. In this chapter we introduce this research problem and its context, and outline our approach to addressing it.

1.1. Adaptive software in an uncertain world

Software systems exist in multi-faceted environments. These environments include the computational and network infrastructure and the resources provided by those infrastructures. They also include aspects of the real world with which the computerised system interacts either by providing or consuming services, or as a controller. There are many sources of change and perturbation in these environments to which a software system may need to adapt. Distributed software systems rely on networks that may have unreliable bandwidth or availability. Mobile systems may have variable access to resources. Heterogeneous systems may need to adapt to new components that use various technologies. Composed software systems are often aggregations of services or black-box components that may be of uncertain behaviour or reliability. Inter-organisational systems may use components that are not directly controlled or owned by the composite system. In open software systems, there may be a need to interact with other systems or agents that exhibit a degree of autonomy from the composed system.

The increasing pervasiveness and integration of software systems within the physical world means that such systems also need to cope with changing requirements. Whether they are personalised software devices, sensing or control systems, or information systems that mirror real-world processes, such software systems often need to cope with changes in user requirements or preferences, changes in the context of use, or changes in the demands of the operating context.

Many types of software system are, therefore, having to cope with simultaneous changes in both requirements and environments, and having to balance both these types of change. For example, in the domain of pervasive mobile computing a user may want to change the device they use (PDA, mobile etc) as they move to a different location. The application may also need to respond to various types of user with different preferences or permissions; or to different personas of the same user; or to a user with changing goals or requirements. Consider a scenario where a user wants to download a picture on to their PDA from a server over the Internet. The user will have certain preferences that need balancing (e.g. speed of download, quality of picture). The PDA also has a display of limited resolution. The bandwidth and server capacity may also be variable. The application responsible for delivering the picture may therefore have to restructure itself by adding a picture compression component to better suit the user's need, the device's capabilities and the bandwidth available.

Given uncertain computing environments such as those described in the above scenario, traditional software development paradigms, that assume static computational environments in a fixed context of use, are inadequate. What is needed is adaptive software that can take account of changing requirements and contexts, and respond accordingly.

1.2. Research problem – an architecture for adaptive software

The aim of this research is to develop an architectural meta-model and framework for the construction of software applications that will be adaptive to both changing requirements and environments. Emmerich (Emmerich, 2000) suggests that distributed architectures need to possess a number of properties. These include scalability, openness, heterogeneity, resource sharing and fault tolerance. In *distributed* systems, it is often considered desirable that such non-functional properties be transparent to the application programmer (for example, middleware could perform load balancing to achieve performance transparency). In *adaptive* systems, on the other hand, non-functional properties may need to be explicitly *managed* and traded-off against each other. In order to effectively implement *adaptive* distributed applications, we contend there are a number of other properties that are desirable in adaptive system architectures.

- *Grounded.* The architecture needs to be able to take account of the 'real-world'. It needs to represent more than just logical relationships between or within abstract computational entities. Software entities and their relationships are always situated in a context which determines many of their non-functional attributes: for example, performance, reliability, security, and so on. Adaptive software needs to be able to take account of (if not model) this situated context by sensing changes, evaluating the effect of those changes and acting on those changes.
- *Exogenous.* The architecture cannot assume it has access to the internal configuration or state of the components that make up the system. Coordination of the behaviour, or measurement of non-functional properties, of the components must be external or exogenous to those components. An exogenous management structure also maintains a separation of management and functional concerns.
- *Self-managed.* It is commonly accepted that parts of a complex system should be able to be independently described, implemented and deployed in modules (Simon, 1969). The complexity of managing the relationships between entities in a system increases dramatically as the number of heterogeneous entities increase. In order to handle this complexity, management of the system also needs to be distributed down to the level of modules rather than globally managed. In other words, these modular composites should be self-managed, as much as possible.
- *Recursive.* If descriptions of the system at different levels of granularity and abstraction are based on the same architectural meta-model, this increases the efficacy of that meta-model and greatly increases the comprehensibility of the design. For example, one of the strengths of object-oriented methodologies is that objects are composed of other objects which are made of other objects, and so on. Units of architectural description should be able to be applied at different levels of granularity.
- *Practical.* The concepts in the architectural meta-model should be as simple as possible (and no more). One of the advantages of the object-oriented methodology is that it is based on a few powerful concepts (encapsulation,

inheritance, polymorphism, etc.) that can be realised in a relatively simple meta-model (class, object, method, etc.). Likewise, an adaptive architectural model should ideally be based on a few powerful concepts that software developers can readily comprehend and apply. The architecture also needs to be able to be supported by frameworks or tools that make it practical to implement industrial scale systems.

Any adaptive architecture will, of course, need management mechanisms for sensing change and adapting to that change. The thesis presents an adaptive software architecture, in the form of a meta-model and an application framework, which has the above properties.

1.3. An organisational approach to adaptive software

One basic approach to building runtime adaptive systems is to construct them of loosely coupled elements. These elements are dynamically coordinated and reconfigured to meet environmental demands or changing goals. This underlying approach is common to autonomic computing, agent systems and dynamic architectures, and is the approach adopted in this thesis.

This thesis presents a meta-model and framework called Role Oriented Adaptive Design (ROAD). The ROAD meta-model is based on the biological concept of ontogenic adaptation. Ontogenically adaptable systems can regulate and restructure themselves while maintaining their organisational integrity. We have taken a system-theoretic approach to developing such systems based on a number of principles. These principles are the separation of control from process (as in a control system), the distribution of control down through a recursive structure, and the radical separation of abstract function (a role) from the implementation of that function (a player). Rather than being monolithic constructions, ROAD software applications are dynamic compositions of distributed, loosely-coupled entities that operate in changing contexts. In this sense they resemble complex human organisations or biological systems, more than they resemble simple deterministic machines.

The ROAD meta-model reflects this organisational approach. The key concepts and characteristics of this meta-model are as follows:

- Software applications are viewed as goal-oriented organisations that attempt to remain viable in changing environments.
- The structure of software organisations comprises a network of functional roles associated by contracts. This structure is a runtime entity rather than just a design-time construct.

- Roles define a purposeful function (fulfil a goal) with respect to the organisation as a whole.
- There is a strict separation of abstract function (role) and process (players). Roles are executed by players (components, services, agents, people, or composite organisations).
- Organisational structures are flexible because the relationships between roles (contracts), and the relationships between roles and their players, can be changed at run-time.
- These flexible organisational role-structures are encapsulated in a composite that is managed by its own organiser. Management includes both (re)configuration and regulation of the composite.
- Because self-managed composites are themselves role players, these composites can be distributed and recursively composed.
- Management is ultimately the management of relationships. Non-functional properties can always be viewed as properties of functional relationships (properties of an entity are always properties *with respect to* some other entity / organisation). In ROAD, therefore, these properties are always defined in, and measured by, contracts.

The concepts in the ROAD meta-model have been expressed in a programming framework which can be extended by the application programmer to create adaptive applications.

1.4. Thesis outline

This thesis is divided into three parts. Part I is a general discussion of the concepts of adaptation and organisation that underlie the ROAD meta-model (Chapter 2), and a review of the literature of adaptive architectures in terms of these concepts (Chapter 3). Part II describes the Role-Oriented Adaptive Design meta-model. Chapter 4 of Part II provides an overview of the meta-model and an expository example. The remaining chapters in Part II describe the concepts in the meta-model in detail: role and players (Chapter 5); contracts (Chapter 6); and self-managed composites, organisers and the management system (Chapter 7). Part III consists of a description of an implementation of the ROAD framework (Chapter 8), a test application built on that framework (Chapter 9), a design case study in the domain of service-oriented computing (Chapter 10), a discussion of the expressiveness of the ROAD meta-model and the efficiency of the framework (Chapter 11), and a conclusion that discusses the contribution of the thesis and points to future work (Chapter 12).

Chapter 2 is a discussion of the system-theoretic concepts underpinning ROAD. This approach is system-theoretic, in that we are concerned with how software systems *as a whole* adapt to changes and maintain their viability in their environments, rather than solely focussing on particular adaptive mechanisms or strategies. We define what we mean by *adaptation.* The focus of this thesis is *ontogenic adaptation* in designed systems. Such systems have three properties: Plastic structure, replaceable elements, and regulatory/managerial mechanisms. We then discuss the concept of *organisation* which is central to understanding ontogenic adaptation, then review key literature on Cybernetics and General Systems Theory to elucidate the related concepts of control, regulation and hierarchy. From these system-theoretic approaches two principles are drawn. The first principle is the separation of control from process. The second is the distribution of control down through a recursive structure. Our approach follows an additional third principle: the radical separation of role (abstract function) from the implementation of that function. We then use these concepts to define a taxonomy of adaptive systems. This taxonomy shows how the type of adaptive software discussed in this thesis fits within the broader context of adaptive systems.

Chapter 3 applies the concept of *ontogenic adaptation*, as discussed in Chapter 2, to software systems, and reviews the literature on adaptive software architectures in terms of this concept. We set out what properties a software system would require for it to be considered ontogenically adaptive in terms of the broad categories of *reconfiguration*, *regulation* and *management.* The existing literature on adaptive architectures is then examined. In order to analyse this literature, we identify a number of distinguishing characteristics of adaptive architectures. After setting the scope of the review and briefly discussing related areas of research not covered, a number of adaptive architectural frameworks are reviewed in more depth, categorising them according to the distinguishing characteristics identified earlier. The chapter concludes by presenting a comparative table that summarises the characteristics of the reviewed frameworks.

Chapter 4 begins Part II of the thesis by discussing the major concepts of the ROAD meta-model, and relating these concepts back to the general properties found, to a greater or lesser extent, in adaptive architectures. The main ROAD concepts are roles, players, contracts, organisers and self-managed composites. We then introduce an expository example that will be used to illustrate these concepts in the subsequent chapters of Part II.

Chapter 5 discusses the characteristics of ROAD organisational *roles* and *players*, and contrasts ROAD roles with other views of roles found in the literature on software design. In ROAD's meta-model, functional roles are runtime entities representing "positions" in an organisational structure. In an organisation, roles have an independent identity and existence from the players who are assigned to them. The radical separation of roles from players raises a number of issues that are addressed. These issues include support for heterogeneous players (objects, agents, services, user interfaces etc.) and players with various levels of autonomy and capability, message queuing and routing in a role structure, and the maintenance of domain state during reconfiguration. In a ROAD application, roles are stateful interfaces that preserve communication states if they have no player attached. Role-players of various capabilities (including humans in some circumstances) can be dynamically assigned to roles as the demands on the system change, or as the environment in which the system operates changes.

Chapter 6 examines *contracts* between roles. In ROAD, contracts are used not only to compose and control associations between roles, but are also used to make role-players 'accountable' for their performance. In other words, ROAD contracts not only define functional relationships, but also define the non-functional properties of those relationships both in terms of requirements (obligations of the parties) and a state-of-fulfilment of those obligations (performance). Such contracts are a way of monitoring and controlling the associations between entities that are loosely coupled. They specify the required performance, and monitor and store actual performance of a role-player with respect to the organisation. In Chapter 6 we also show how patterns of interaction in a contract can be abstracted into, what we call, *performative* contracts that represent the authority relationships between two roles. Performance measurement points that correspond to various synchronisation approaches are also defined at this abstract performative level.

Chapter 7 discusses how *self-managed composites* can be created from roles, players, contracts and organisers. We begin by describing the properties of these composites. Issues of message delegation, player containment and whether or not composites perform any domain process or maintain state are also discussed. We then describe the function of the composite *organiser* who is responsible for managing the composite. The functions of an organiser *role* are distinguished from that of an organiser *player*. Organiser roles define *how* to change the composite, while organiser players *decide what* to change. An example of an organiser-player strategy for adaptation *within* a self-managed composite is described. The focus of the chapter then turns to the *management system* which connects the organisers *across* composites. This management system is distinct from the functional system comprising functional roles and players. Adaptive behaviour *across* composites is described including the propagation of non-functional requirements and performance information. The conclusion to the chapter includes a summary model of all the main ROAD concepts presented in Part II of the thesis.

Chapter 8, the first chapter in Part III of the thesis describes an implementation of the ROAD meta-model in the form of a framework that can be extended by the application programmer. We then give an overview of how the concepts, described in the previous part of the thesis, map to key abstract classes in the framework. The subsequent sections describe in more detail the Java implementation of these key ROAD concepts; namely, roles, contracts, self-managed composites and organisers. In particular, we focus on a novel use of instantiable aspects called "association-aspects" to implement contracts at the performative and functional levels. The chapter concludes with a discussion of the work that could be done to further develop the ROAD framework, and what tool support is needed by developers to make practical the development of adaptive software organisations.

Chapter 9 describes the implementation of a test application based on the ROAD framework. The application demonstrates the various capabilities of the ROAD framework, shows how the composite is created using roles, players and contracts, and shows how the framework prevents unauthorised communication. The application also demonstrates the adaptive behaviour that the ROAD framework facilitates by showing how the organiser attempts to mitigate underperformance by reconfiguring the composite. The chapter concludes by showing how a ROAD application can work with heterogeneous players – in this case Java objects and Web services.

Chapter 10 presents a case study that highlights the way ROAD can be used to create Web service compositions, in particular the mediation between changing requirements and the changing provision of services. A number of other modelling capabilities of ROAD are also demonstrated including types of abstract performative contract suitable to inter-organisational contracts (e.g. buyer-seller contracts as distinct from intra-organisational contracts such as supervisor-subordinate, peer-peer, etc.); the use of contracts which govern 'long-lived' transactions; the ability to represent 'virtual enterprises'; and the representation of performance in terms of non-temporal utility. The case study also illustrates the design of composites so that role abstractions in a composite are always kept at the one level of abstraction. Roles are not decomposed into other roles within a composite. Rather, roles are always played by loosely coupled players, some of which may be role composites. Highly adaptive systems can thus be created, because the decomposition (not just the configuration) can always be changed at runtime.

Chapter 11 evaluates the expressiveness of the ROAD meta-model and the performance of the ROAD framework. We first evaluate how well the ROAD meta-model expresses those qualities necessary in an ontogenically adaptive software system as set out in Chapter 3. The chapter then analyses the prototype implementation of the ROAD framework in terms of the runtime overhead it imposes. ROAD defines an organisational middleware structure through which passes all communication between the application's functional runtime entities. This interposed message-intercepting structure creates an overhead compared to, say, the communication between two directly communicating objects. The run-time performance overhead of a ROAD application therefore needs to be characterised relative to such direct communication. We also compare the overhead imposed by ROAD middleware to the overhead imposed by the predominant middleware for more open inter-organisational application integration, namely Web services.

Chapter 12 concludes the thesis with a discussion of its contributions to the field of adaptive software architectures, and then outlines the future work that could be done to further develop the ROAD approach to developing ontogenically adaptive software.

The reader may note that there is no chapter called "Literature Review" in this thesis. This is due to the breadth of the topic of the thesis and its eclectic genesis. The thesis draws from literature on systems theory, adaptive software architectures, roles, and contracts, as well as research on aspect-oriented programming. Although the major review of literature is the review of adaptive software architectures in Chapter 3, discussion of other literature related to the various concepts and technologies can be found in the relevant chapters.

2 Adaptive Organisations

This chapter discusses the concepts that underpin the ROAD (Role Oriented Adaptive Design) approach to developing adaptive software systems. This approach is system-theoretic, in that we are concerned with how software systems *as a whole* adapt to changes and maintain their viability in their environments, rather than solely focussing on particular adaptive mechanisms or strategies. The discussion in this chapter focuses on adaptive systems in general, and provides a context for the literature review of adaptive software systems in Chapter 3.

The first section defines what we mean by *adaptation*. We adopt, from biology, the distinction between three fundamental types of adaptation, namely: evolutionary, ontogenic and environmental manipulation. The focus of this thesis is ontogenic adaptation, and we extend this concept to designed goal-oriented systems.

Section 2.2 of this chapter focuses on the concept of *organisation* which is central to understanding ontogenic adaptation in living and artificial systems. We define what we mean by *organisation* highlighting the difference between emergent natural systems and goal-directed designed systems. We then examine some of the key system-theoretic literature to eludicate the concepts of control and structure in organisations. In particular, Beer's Viable System Model (VSM) (Beer, 1984; Beer, 1985; Beer, 1979) is discussed as a model that draws on these system-theoretic views of control and structure. VSM presents a control-theoretic approach to recursively structured adaptive systems. In particular, VSM differentiates various types of control. It also shows how control can be related to organisational structure, and how structure can be used to manage complexity.

While we do not follow the VSM architecture in this thesis, it does illustrate two important principles for *designed* goal-oriented adaptive systems. The first principle is the separation of control from process. The second is the distribution of control down through a recursive structure. In Section 2.3 we propose a third principle, and one not found in VSM, namely the radical separation of role (abstract function) from the implementation of that function. This provides a plastic flexible structure between loosely-coupled elements that makes adaptation possible.

These concepts are then used to define a taxonomy of adaptive systems in Section 2.4. This taxonomy is presented in order to show how the type of adaptive software we discuss in this thesis fits within the broader context of adaptive systems.

2.1. Adaptation

Adaptation is a relationship between a system and its environment where change is initiated to facilitate the survival of the system (or system type) in that environment. This definition, however, does not illuminate a number of important aspects of adaptation. To more fully characterise adaptation in a system the following questions need to be answered.

- What is the *goal* of adaptation – is the goal just to ensure the survival of the system within its environment, or does the system also have to meet *first-order*[1] goals?
- What *causes* the need for change in the system – is it a change in the goals or requirements of the system, or is the system responding to environmental perturbation?
- What *aspects/qualities/parts* in the system are subject to change and what aspects remain invariant?
- What are the *limits* to adaptation? Every system, living or designed, exists in an environmental context and all adaptation is limited. Systems are not in themselves adaptable – they are adaptable with respect to a set of environmental states. Even systems that we regard as highly adaptive (such as humans) are only viable within a limited range of environmental conditions (atmospheric composition, temperature etc.) and within specific ecology.
- To what extent can a system cope with *unanticipated* changes to the environment or its goals? In other words, do the requirements and environmental states to

[1] Skyttner (2001p80) cites Deutsch's hierarchy of goal-seeking. First order goal-seeking is related to immediate satisfaction or reward; second order goal-seeking achieves first-order goals through self-preservation; third order goal-seeking relates to the preservation of the group; and fourth order goal-seeking preserves the environment or ecosystem. To be *viable* in the long-term, systems need to take account of all these types of goal.

which the system can adapt need to be defined when the system is reified, or can the system dynamically adapt to unanticipated states provided certain limits or constraints are not violated? Can the system *change itself* (structural adaptation rather than just behavioural adaptation) to cope with unanticipated change?

- Can the system *change the environment*? In designed software systems we draw a boundary between the system and the environment, and tend to assume the environment cannot be changed. However, all systems effect their environment to some degree. Adaptation expresses a relationship between a system and its environment. For example, as human beings, it is our ability to modify our environment that has made us so adaptable.
- Is the *agency* for adaptation internal or external to the system? Artificial systems are often classified as *adaptable* (able to be modified by an external agent) and/or *adaptive* (able to change itself).

In order to better build adaptive systems we need a more refined understanding of the nature of adaptation. Biological systems have been a source of inspiration for the development of adaptive software systems. In biological systems two mechanisms of adaptation are commonly characterised (Maturana and Varela, 1980) — evolutionary (phylogenetic) and ontogenic (or ontogenetic) adaptation. If we define adaptation as the compatibility between a system and its environment we can add a third category — environmental manipulation. Each of these mechanisms has parallels in designed systems. These three mechanisms are based respectively on *reproduction*, *self-production* and *production*.

2.1.1. Evolutionary adaptation (reproduction)

Evolutionary (phylogenetic) adaptation is a selective mechanism whereby instances of a class of system *reproduce* themselves with variations. The variants that have a better fit with their environment are selected because they can survive and reproduce. Evolutionary adaptation is, therefore, the adaptation of a species (phylogeny) rather than an individual of that species. In biological systems this variation is random ("blind variation") and environmental conditions determine selection. In software systems, evolutionary adaptation has analogies at both design-time and runtime. At design time the versions of a product throughout the software development life-cycle could be regarded as variations. Unlike biological systems, these variations are not "blind". Versions of the software product-line are based on models the designers hold and evolve in the iterative development cycle. Versions are tested to ensure they meet requirements (i.e. they are well adapted to the nominal environment). On the other hand, genetic algorithms (Holland, 1992) use blind variation and selection at runtime to achieve this type of adaptation. Evolutionary adaptation is based on the *reproduction* of individuals.

2.1.2. Ontogenic adaptation (self-production)

Ontogeny is the history of structural transformations of an individual system. *Ontogenic (or ontogenetic) adaptation* is the ability of a system to regulate itself and change its structure as it interacts with the environment. This change of structure is of two types. The first is the change or interchange of the elements within the structure. In biological systems an example of this type of change is the death and replacement of cells. In software or hardware systems an analogy would be the replacement of one component by another component, where both components share a common interface. The second type of change is modification of relationships between the elements that make up the system. In animals this *plasticity* of structure is achieved, in part, by the nervous system. The nervous system (including the brain) enables and modulates the interactions between various components of the living system by continually modifying itself. This malleability of the nervous system (the individual self-structuring to fit the environment) is the basis of cognition, learning and social behaviour (Maturana and Varela, 1987).

In biological systems, the plasticity of structure that enables ontogenic adaptation has arisen through evolutionary adaptation. In software systems we must design this plasticity (indirection) into the system. As with biological systems, there are two types of indirection, firstly the ability to interchange elements that fulfil the same function within the structure, and secondly the changing of the relationship between those elements. In the context of software systems, we call these two types, respectively, *indirection of instantiation* and *indirection of association* (Colman and Han, 2005).

As well as having (at least some) loosely coupled elements, an ontogenically adaptive system must have ways of *determining* at runtime that indirection (i.e. deciding what the structure, and behaviour over that structure, will be), so that the system maintains its ***viability***. A viable system[2] is one that can continue to survive (and, if required, continue to meet its goals) in uncertain and changing environments. Maintaining viability may involve the system learning either in a representational or non-representational form. Neural networks are examples of systems that exhibit ontogenic adaptation through non-representational learning. On the other hand, in the

[2] Our use of the term *viability* derives from the Viable System Model of Stafford Beer (Beer, 1984) who applied concepts of cybernetic control to human organisations.

case of systems that hold a reflective representation of themselves, maintaining such viability may be the responsibility of a management system.

Ontogenic adaptation can be viewed as a form of *self-production* (autopoiesis[3]) (Maturana and Varela, 1980) — the ability of the system to maintain its ***organisational integrity*** even through the elements within it may change and the relationships between those elements change.

2.1.3. Environmental manipulation (production)

There is a third type of adaptation – the ability to change the environmental constraints to suit the system. This type of adaptation is also apparent in biological systems. Biological examples include symbiosis; ant pheromone trails to food sources; the creation of nests to moderate environmental perturbations; agriculture; and culture. In software engineering the co-evolution of a system and its environment has long been recognised. The design of software systems has been characterised as interactions between "system designers" and "environmental designers" (Sykes, 2003). In software systems, the boundary between system and environment is not just a technical issue. Where this boundary lies, and whether or not the environment can be changed, is often a negotiated socio-technical decision. As software systems are developed for more complex and open software environments, the manipulation of environmental constraints may become an important form of adaptation. An example of such environmental alteration is the emergence of social norms in multi-agent societies (Zambonelli, Jennings and Wooldridge, 2000). Adaptation through environmental manipulation can be thought of as *production*.

2.2. Organisations

Adaptation through evolution, ontogenesis, and environmental manipulation are complementary, and in many real world systems occur simultaneously. In this thesis we will focus on ontogenic adaptation of software systems. In this section we examine the relationship between organisation and ontogenic adaptation.

2.2.1. Ontogenically adaptive systems

. We can summarise the general properties of an ontogenically adaptive system as follows:

- Flexibility (plasticity) of structure (indirection of association)

[3] Maturana and Varela see the autopoiesis as *the* defining characteristic of living systems. The systems we discuss in this thesis are, of course, not living. This is, in part, because they are not entirely self-producing. Artificial systems have designers at some stage in their life-cycle. However, such artificial systems can still have ontogenically adaptive properties.

- Replaceable elements (indirection of instantiation)
- Ability to 'manage' (regulate and reconfigure) the indirection in the system to ensure its organisational integrity and thus its viability in its environment.

As pointed out above, it is the *organisational integrity* of an ontogenically adaptive system that is invariant in dynamic situations, rather than the elements or structure of the system. However, while the concept of *indirection* is clearly understood in the context of software systems, applying the biologically inspired notions such as *viability* and *organisational integrity* to software systems is a more problematic task. If we are to build ontogenically adaptable software that can maintain, at runtime, its organisational integrity, we will need to define what organisation is, how it might be represented, and how the organisational structure might be manipulated to achieve ontogenic adaptation.

As software systems become more complex and operate in more open and uncertain environments, we argue that it is necessary to model the dynamics of those systems at a higher level of abstraction – at an organisational level rather than just a structural, functional or process level. For example, biological science describes the complex regulatory mechanisms that keep living organisms in a homeostatic relationship with their environment (including other organisms). Similarly, management theory has much to say about the design of organisational structures in human organisations so that these organisations can thrive within their business environments, and be more responsive to changing goals (e.g. (Mintzberg, 1983; Simon, 1957)). In this work, we aim to show how software systems can be represented at an organisational level of abstraction, so that we can start to reason about software organisation and, thus, build viable ontogenically adaptive systems.

2.2.2. Organisation as metric, process, structure and system invariant

A description of a system's *organisation* is a description of the relationships between elements in that system. There are many definitions of organisation. Parunak and Brueckner (Parunak and Brueckner, 2003) suggest three aspects of organisation based on information entropy, process, and (emergent) structure. Organisation$_1$ (O_1) is a *measurement* of *state*; that is, the inverse to the amount of entropy in the system based on some regularity — spatial, functional or temporal. O_2 is a *process* in which O_1 increases in time, in other words the process that decreases the amount of entropy within the system. Finally, O_3 is the *structure* resulting from O_2 which can be measured with O_1.

This definition nicely binds the physical and informational aspects of organisation together but it has two shortcomings. Firstly the measure of organisation (O_1) is itself a state-based metric. A more system-theoretic definition of organisation can be found in Maturana and Varela (Maturana and Varela, 1980). In their definition, organisation is the set of relationships that maintain the viability of a complex biological system in a changing environment. In complex systems these associations can be seen as *defining* the system. For example, in a complex multi-cellular system such as an animal, cells are continually dying and being replaced. What stays constant in such biological systems is the functional relationships between the roles played by these cells. It is these relationships that define a system as a unity, and determine the dynamics of interaction and transformations (the ontogenic adaptation) which the unity may undergo (Maturana and Varela, 1987). Organisation is what maintains the system as a *viable* entity in a changing environment.

The second limitation, for our purposes, of the definition in (Parunak and Brueckner, 2003) is that it only addresses *emergent* structures such as those in natural systems. In terms of adaptation, natural systems only need to ensure their survival. On the other hand, software systems need to survive but they also are *designed* to achieve goals. The above definition, therefore, needs to be modified to take account of the organisation's purpose. The process (O_2) can include activities to deliberately modify the structure (O_3) to achieve the system's purpose. In the next sections we examine system-theoretic approaches that account for such goal-orientation in adaptive systems.

2.2.3. System-theoretic approaches to adaptation and organisation

To remain viable in their environment, adaptive systems need to have a goal of achieving a stable state with their dynamic environment. Such *homeostatic* systems dynamically self-regulate in order to maintain certain variables within acceptable bounds (e.g. the human body regulates its processes in order to keep blood temperature constant). Ashby (Ashby, 1956) expresses the interdependence in the dynamics of a homeostatic system and its environment in his concept of *ultrastability.* Ultrastability is the ability of a homeostatic system *to change its internal organisation or structure in response to external conditions*. Such stability is achieved through control feedback loops. The existence of such regulatory mechanisms is an essential feature of all adaptive systems, and one of the fundamental concepts in cybernetics.

Cybernetics is the attempt to develop a "science of control and communication, in the animal and the machine" (Wiener, 1961). In this sense cybernetics is a generalisation and extension of control theory. Key to the cybernetic view of control is the separation of a controller from the process being controlled, as illustrated in Figure 2-3 below. The process interacts directly with the environment, while information on the state of the environment and the process flows to the controller to form control loops. These control loops can either be *feedback* or *feed-forward*. Feedback (or closed-loop) control adjusts the process based on the measurement of some *controlled variable* that is affected by the process in the environment. A simple feedback control loop is illustrated in Figure 2-1 below (adapted from (Shaw, 1995)). The desired state of the system (set point goal) is compared to the actual state, as measured by some control variable. A manipulated variable is then altered to decrease the difference between the desired and actual states. Changes to which a system must adapt are illustrated by block arrows.

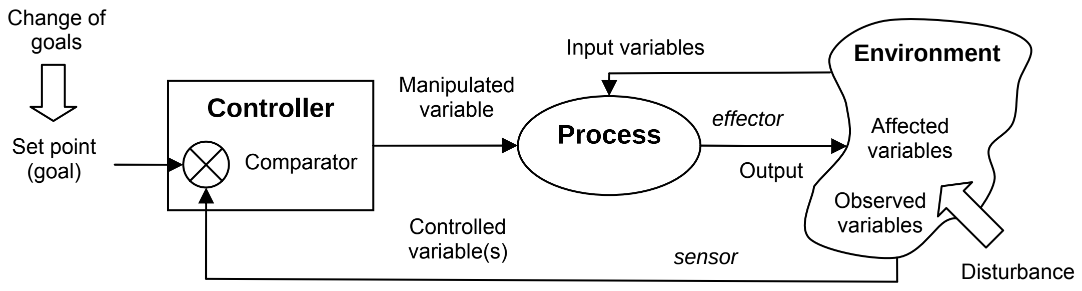


Figure 2-1: Feedback Control

In feed-forward control, on the other hand, the controller must *anticipate* the effects on the controlled variable resulting from changes to the manipulated variable, given various environmental states. To do this the controller must maintain a dynamic model of the controlled system in its environment. In general, a controller must be able to respond appropriately to all possible states of the environment, that is the *variety* of the environment. Ashby's *Law of Requisite Variety* states that to regulate a system, the controller must be capable of generating at least as much variety as that exhibited by the system being regulated.

Like cybernetics, General Systems Theory (GST) (Bertalanffy, 1968; Skyttner, 2001), attempts to distil general principles of biological and social systems. Such systems are viewed as a "system of systems" with each subsystem performing a function in the enclosing system. Living organisms have evolved from homeostatic couplings of self-regulating systems that create composed systems at a higher level. These systems in turn compose other systems of ever increasing complexity. Systems theorists view this hierarchical composition as an essential feature of all complex adaptive systems. A number of hierarchical system models have been developed (see Skyttner, 2001) that attempt to extend the principles to complex man-made systems.

These include the Viable System Model (Beer, 1985) that is discussed in the next section.

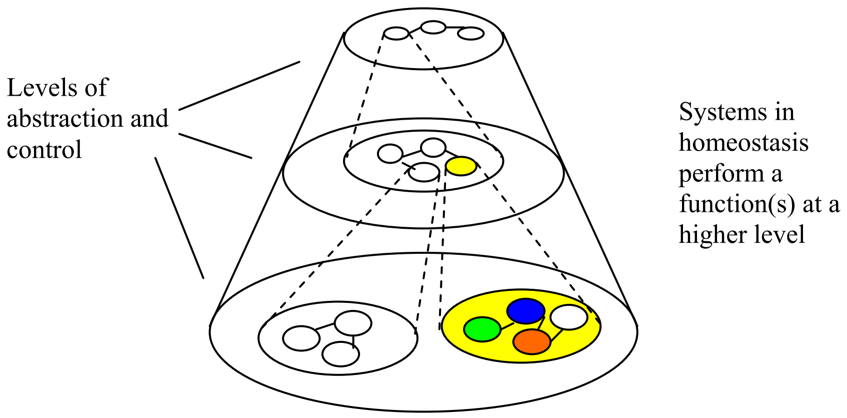

Levels of abstraction and control
Systems in homeostasis perform a function(s) at a higher level

Figure 2-2: A multilevel “system of systems” hierarchy

Heylighen & Joslyn (2001) nicely summarise this hierarchy of systems from a cybernetic perspective: “A control loop will reduce the variety of perturbations, but it will in general not be able to eliminate all variation. Adding a control loop on top of the original loop may eliminate the residual variety, but if that is not sufficient, another hierarchical level may be needed. The required number of levels therefore depends on the regulatory ability of the individual control loops: the weaker that ability, the more hierarchy is needed. This is Aulin's *law of requisite hierarchy*”. A similar point has been made by Herbert Simon in his study of human organisations (Simon, 1957); namely, that hierarchy in artificial organisations compensates for the “limited rationality” of the members of that organisation[4]. In his comparison of the natural and artificial worlds Simon (Simon, 1969) sees quasi-autonomy from the outer environment as an essential characteristic of complex systems. Complex systems are aggregations of “stable intermediate forms”. Designed goal-oriented systems can also be viewed as a hierarchy of control where successively lower levels of the system operationalise the goals of the next higher level (e.g. a bureaucracy with various levels of management).

[4] Agre (1995) notes that Simon (1957) outlined many ways in which social organisations compensate for the "limited rationality" of their members. “The orchestration of numerous workers within a larger organization, Simon argued, compensates for the individual's limited capacity for work. Likewise, the division of labor and the assignment of specialized tasks to individuals compensates for their limited abilities to learn new tasks. The flow of structured information through the organization compensates for their limited knowledge, and the precise formats of that organization, together with the precise definition of individual tasks, compensate for individuals' limited abilities to absorb information and apply it usefully in making decisions. Finally, Simon believed that the hierarchical structure of bureaucracies compensates for individuals' limited abilities to adopt their own values and goals.”

In these goal-directed systems, control is distributed down through the structure to quasi-autonomous subsystems.

In living systems the boundaries of these "intermediate forms" are internally determined (Maturana and Varela, 1980). The cell membrane is the archetypal example of such a boundary. As we have already discussed in Section 2.1.3, in artificial systems the boundaries of the system is often arbitrarily determined rather than internally determined. However, even artificial organisations typically have a well defined interface or 'membrane' which regulates its interactions with the environment.

2.2.4. The Viable System Model – an example of combining control and structure in a goal-driven system

Other authors such as Beer (Beer, 1979; Beer, 1984), have applied cybernetic principles to goal-driven business organisations and bureaucracies. Such *intentionally designed* adaptive systems may need to adapt to changes in the goals of the system, as well as adapting to changes in itself or its environment. To do this they need to regulate their internal interactions and their interactions with their environment. Beer's Viable Systems Model (VSM) is a management theory but it has also been applied to many forms of biological and social organisation. VSM combines the cybernetic concepts of balancing variety, with a General Systems Theory (GST) "system of systems" approach to creating a hierarchy of self-regulating "viable" systems.

The strength of VSM is that it differentiates a number of types of management control, and provides a consistent framework of how these types interact within and between viable systems. All viable systems have "five necessary and sufficient subsystems interactively involved in any organism or organisation that is capable of maintaining its identity independently of other such organisms within a shared environment." (Beer, 1984). A viable system is separated into a System ONE[5] and Control Systems (TWO-FIVE) as illustrated in Figure 2-3 below. This distinction between *process* and *control* is at the heart of Beer's cybernetic approach to organisation. In a viable system, the variety (complexity) of the control system and the process, and the environment, must be kept in balance (homeostatic equilibrium) through amplification and attenuation of complexity in order to preserve Ashby's *Law of Requisite Variety*.

[5] Beer's use of the word "System" is somewhat confusing here. CAPITAL letters are used to denote these VSM systems. Only System ONE is a system in the sense of an independent entity. Systems TWO to FIVE can be thought of as management subsystems that interact with each other, and with other subsystems of the same type in other System ONEs. For example, the Planning and Adaptation subsystems (System FOUR) in nested systems have connections to each other.

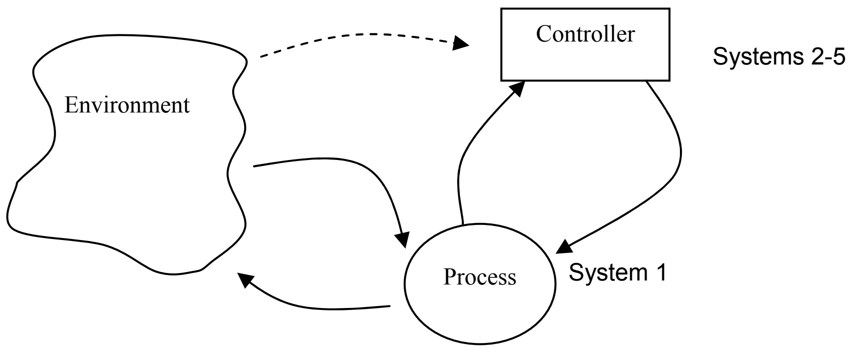


Figure 2-3: Separation of Control from Process

System ONE (S1) in VSM terminology is the process or sub-system being controlled. This process enacts the primary function or purpose, and is itself a composite of viable systems. The management systems that make up the controller are as follows:

- System TWO (S2) – Regulation, coordination (e.g. production relations). S2 damps the oscillations between S1 and its environment.
- System THREE (S3) – Command, control (e.g. line management) and auditing functions
- System FOUR (S4) – Adaptation, planning, strategy, simulation. To make the system adaptive to non-routine change, S4 needs to be aware of the environment, and dynamically adapt to unanticipated changes in the environment. To do this it needs a model of the system and its environment
- System FIVE (S5) – Supervisory control that defines goals and policies for the system, and gives identity to the system

These management sub-systems exemplify the hierarchy of control loops discussed in the previous section. The relationship between these VSM systems is illustrated in Figure 2-4 below. Figure 2-4 also illustrates that there may be more than one S1 (in the diagram Systems 1A and 1B) which are coordinated hierarchically through S3. As well as interconnected subsystems existing at the same level, all viable systems are recursive; any viable system contains, and is contained in, a viable system.

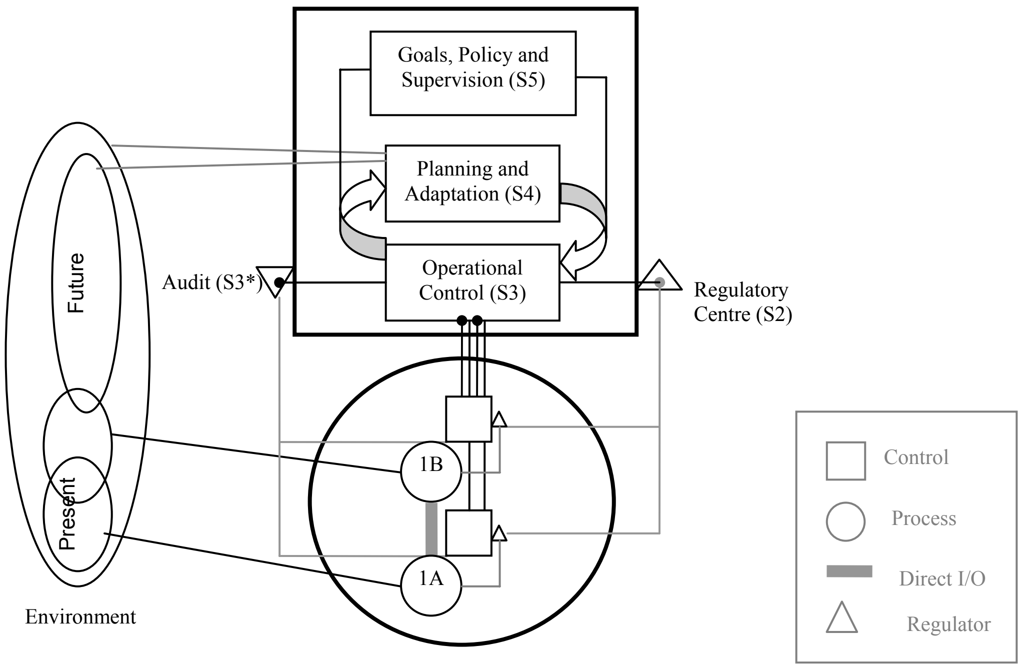


Figure 2-4: A Schema of a Viable System (adapted from Herring (2002))

In summary, Beer's Viable System Model combines the control-theoretic perspective of cybernetics, with GST's view of complex systems as a recursive hierarchy of systems. VSM recognises the importance of different types of control in viable systems, namely regulatory, operational, adaptive and supervisory control. These nested control loops attempt to ensure that the system stays in a stable state in the face of environmental perturbation and shifting goals. VSM requires the differentiation and implementation of these various control mechanisms. The hierarchical nature of VSM means that these control mechanisms are applied at all levels of the system. By localising the management to the appropriate level, VSM provides a framework for controlling complexity, by limiting the amount of variety with which any one controller has to cope.

However, creating an ontogenically adaptive system by separating control from process (as in VSM), poses some challenges. Controllers must maintain a model of the process (or at least those parts that need to be controlled and adapted). This model must be valid and be kept updated. In a complex system, this self-model can add considerable complexity to the system assuming a valid model can be defined at all[6]. While cybernetics, GST and VSM in particular provide insights into the nature of hierarchy and control in complex adaptive systems, the very generality of these

[6] On the other hand, self-organising systems that evolve adaptive strategies through evolution (CAS systems such as ant colonies) do not require a separation of control and process and do not have to maintain explicit models. While this manages the demon of complexity, evolutionary adapted systems are inherently brittle, do not have first-order goals, and are not amenable to supervisory control.

approaches is also their shortcoming[7]. There is little evidence they have had an impact, as yet, on informing the design of real software systems although preliminary attempts have been made in applying VSM to software design (Herring, 2002; Cai, Cangussu, DeCarlo et al., 2004). Herring's extension of VSM to software systems is discussed in the next chapter. Although the VSM approach is a promising approach for creating adaptive systems control and recursive composition, its quest for generality blurs a fundamental distinction between naturally evolved and intentionally designed systems. In designed systems we can radically separate the teleological function (role) of a system from the implementation of the system.

2.3. Role-based organisations

In addition to the system-theoretic principles described above (namely, the separation of process and control, and the distribution of control through a recursive structure) this thesis introduces a third principle for goal-oriented ontogenically adaptive systems: the separation of organisational roles from the players that execute those roles. A role represents the abstract function[8] of an entity within an enclosing system. In natural systems this abstract function defines a *teleonomical* 'purpose' that is *ascribed* by the observer (e.g. "the heart's function is to circulate blood around the body"). In designed systems, on the other hand, the function of the system/subsystem is a *teleological* purpose that initially exists in the mind of the designer. The designer can envisage the *role* of the system/subsystem without specifying how that system will be implemented. This separation of abstract definition and implementation is commonplace in software systems (e.g. interface of an object versus the implementation of the object) and in human organisations (e.g. a job position is filled by an employee).

In designed systems *organisational descriptions* are means-end functional descriptions and are at a higher level of abstraction than perspectives based solely on descriptions of either structure or process. For example, in object-oriented design these structural and process perspectives are captured, respectively, in class and collaboration diagrams. Neither of these representations captures the *purpose* of the entities

[7] We might speculate that the messy real world does not conform to the discrete states required by cybernetics. In particular there is a disjunction between the discrete states and control variables we can define in artificial systems, and the complex interdependencies of the analogue real world. Ignoring the challenges imposed by this chasm has led to exaggerated claims and subsequent lack of outcomes from cybernetics and GST. By confusing information-flow with structural-coupling (Maturana and Varela, 1987), more extreme expressions of "second order" cybernetics and Information Theory (Shannon and Weaver, 1949) have conflated the representation of the system with the system itself.

[8] The word *function* is highly ambiguous. Nagel (1961 p256) lists six distinct usages of the word. We use the word here in its biological sense, i.e. a functional explanation explains the behaviour of a system in terms of its purpose or Aristotelian end-cause. Such explanations are common in physiology, biology and the social sciences. The word *function* as commonly used in mathematics and computing, meaning a mapping or transformation, lacks this teleological aspect.

represented. The concept of a role, on the other hand, *does* capture the purpose of the entity. The shortcoming of many software descriptions is that they reduce the description of organisation to just the topological structure or to just the process/behaviour. A complete organisational description would need to indicate how goals are *transmitted* through the system; how the entities are *coordinated* to avoid performing extraneous or mutually destructive activity; how the system *changes* in response to changing goals and environmental perturbations; and how the system maintains its organisational viability.

2.3.1. Organisational structures

A role within an organisation satisfies responsibilities to the system as a whole. *Roles are the nodes of designed organisational structures*. This view of organisation has much in common with the conception of human organisations (e.g. bureaucracies) where people fill positions (roles) within an organisational structure. An organisational structure can be described separately from the players who perform those roles. In our approach we distinguish between *functional* roles that fulfil some domain function (at the process level S1 in VSM terms), and *management* ("organiser") roles that regulate and adapt the system (S2-S5 in VSM terms). We will discuss roles in more depth in Chapter 5.

As defined above, ontogenic adaptation is the ability of a system to change its structure as it interacts with the environment. The separation of roles from their players provides one degree of freedom in a role-based organisation, which we call *indirection of instantiation*. Another degree of freedom in role-based organisations is the ability to alter the relationship between roles in that organisation. Altering these associations changes the organisational structure itself. We call this flexibility *indirection of association* (Colman and Han, 2005). These two types of indirection are illustrated in Figure 2-5 below.

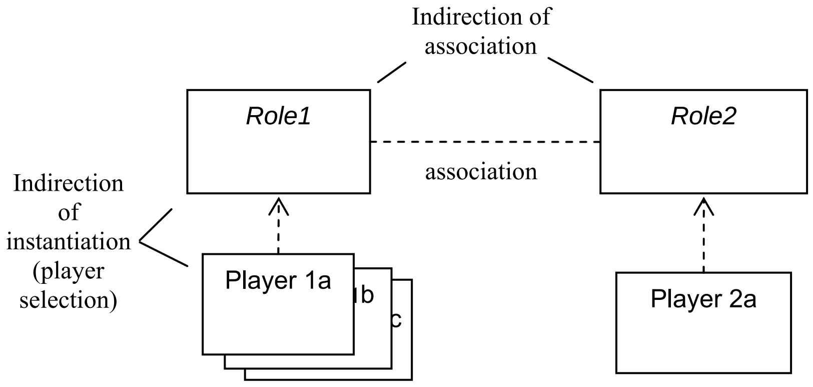

Indirection of association
Role1
Role2
association
Indirection of instantiation (player selection)
Player 1a
b
c
Player 2a

Figure 2-5: Two dimensions of indirection in adaptive role-based organisations.

The network of associations between roles forms an organisational structure. In ROAD, these associations are created using contracts. ROAD contracts are discussed in Chapter 6.

2.3.2. Management of flexible role-based structures

The design of organisations involves the division of function into roles (Mintzberg, 1983). However, while this near-decomposability (Simon, 1969) of function in goal-oriented systems may provide us with an organisational structure, it does not tell us how that structure is coordinated or managed. In a live system any indirection must be determined before or during runtime. An adaptive composite system needs to *manage* its indirection. In this thesis we follow the approach of VSM and treat control/coordination/management as a separate subsystem(s) from the functional system (System ONE in VSM terms). This coordination-system can be described and controlled independently from the functional subsystems that interact directly with the application domain. This approach is analogous to the coordination-systems that exist both in living things and in man-made organisations. In the realm of biology, the autonomic nervous system can be viewed as a system that, in part, coordinates the respiratory, circulatory, and digestive systems. Similarly, the management structure or financial system in a manufacturing business can also be described at a separate level of abstraction from the functional processes that transform labour and material into products.

This management involves both the ontogenic *configuration* (composition or reconfiguration[9]) of the system, and the *regulation* of interactions over the composed structure. To *configure* the structure the management system needs to be able to dynamically create bindings between its loosely coupled roles and players on both the above dimensions of indirection in response to changing demands and the changing environment. To *regulate* the performance of the composed structure, the management system needs to maintain some form of representation of the *requirements* and *current state* of the underlying functional system. In cybernetic terms, these are *control variables* as in Figure 2-1 above. These models will vary depending on the variables that need to be controlled in order to maintain the system's viability in its environment. A biological example of a controlled variable is the level of oxygen supply to the cells. In a business, the variable might be the amount of funds in the bank. In computerised systems, such control variables could be derived from utility functions that measure

[9] We use the word *configuration* to include both the concept of *composition* from scratch and the *reconfiguration* of an existing structure.

computational or communication resources; or variables in the environment with which the system interacts. Management of software role structures is discussed in Chapter 7.

2.4. Summary - a taxonomy of adaptation in systems

We can summarise the above discussion by presenting a taxonomy of adaptation in systems. There a many forms of adaptation in systems, and in this thesis we only address a subset of those forms. In Figure 2-6 below, the boxes in bold represent the focus of this thesis.

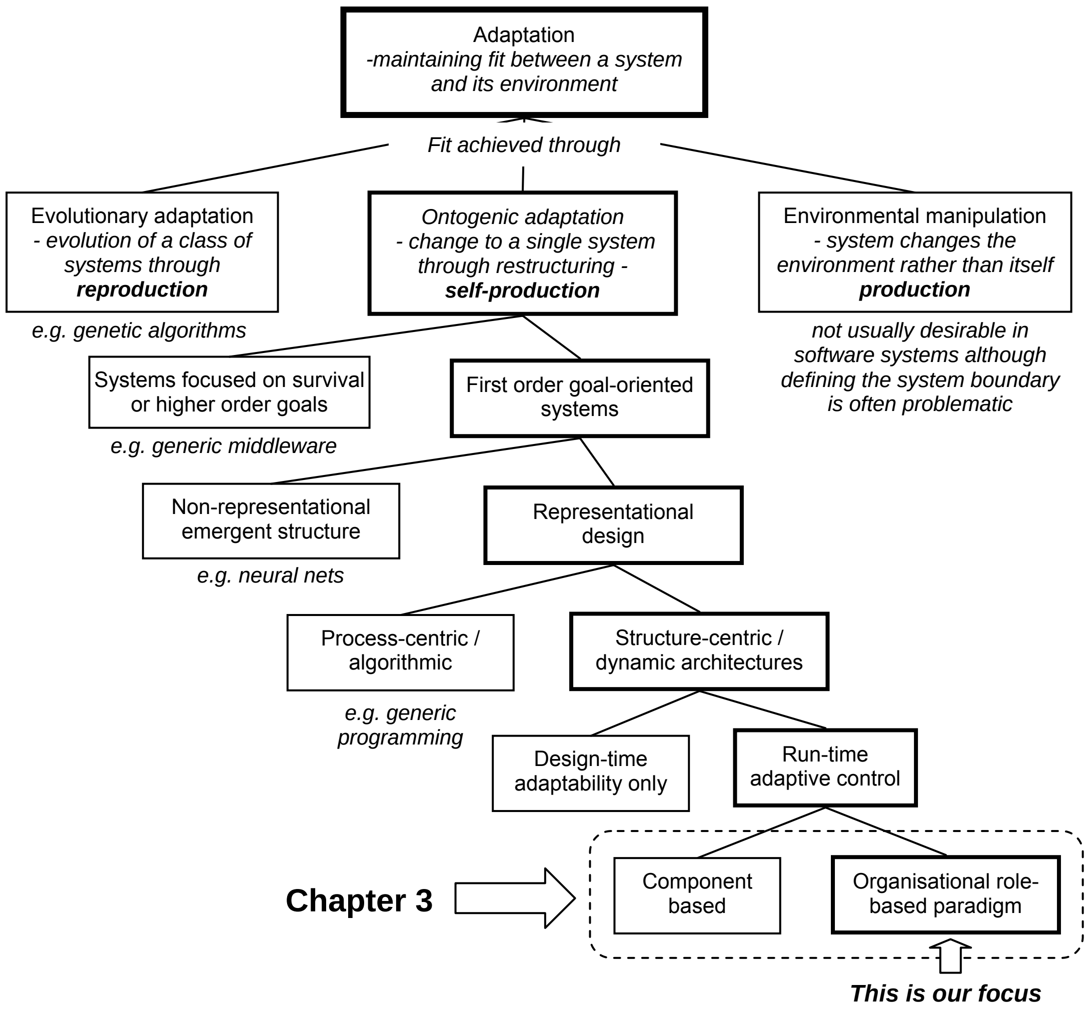

Adaptation
-maintaining fit between a system and its environment
Fit achieved through
Evolutionary adaptation - evolution of a class of systems through reproduction
e.g. genetic algorithms
Ontogenic adaptation - change to a single system through restructuring - self-production
Environmental manipulation - system changes the environment rather than itself production
not usually desirable in software systems although defining the system boundary is often problematic
Systems focused on survival or higher order goals
e.g. generic middleware
First order goal-oriented systems
Non-representational emergent structure
e.g. neural nets
Representational design
Process-centric / algorithmic
e.g. generic programming
Structure-centric / dynamic architectures
Design-time adaptability only
Run-time adaptive control
Chapter 3
Component based
Organisational role-based paradigm
This is our focus

Figure 2-6: Taxonomy of adaptation in systems

Reading from the top of the taxonomy in Figure 2-6, the quality of adaptability in a system is the maintenance of a fit between the system and its environment. Fit means the ability of the system to survive or remain viable within its environment. There are three classes of adaptation: evolutionary, ontogenic and environmental manipulation. These classes alter, respectively, the system type, the individual system and the system's environment. In this thesis we are concerned with ontogenic adaptation – the restructuring of the individual system.

Systems that exhibit ontogenic adaptation can be further divided between those that are only focused on survival, and those that *also* need to meet first-order goals, i.e. that are teleological. Natural systems are typically thought of as being focused on survival, whereas artificial systems, such as software systems, are designed to maintain or achieve first-order goals.

Goal-orientation in ontogenic systems can be the *emergent* behaviour that is a result of processes such as reinforcement learning, or can be a result of the *deliberative design* of structures that the meet those goals. In the former case the structure of the organisation is emergent (Holland, 1998). In the software domain, emergent structure is typified by neural networks and swarm intelligence. While such systems, such as ant-colony simulations, can be trained or tweaked to achieve particular goals in environments with limited dynamic variation, they tend to be focused on solving particular classes of problem (e.g. pattern recognition, search algorithms), and are not adaptable to changing goals[10]. On the other hand, this thesis is concerned with systems whose structures are deliberately designed to facilitate the operationalisation of first-order goals.

As such the approach here is structure/architecture centric. This is in contrast to algorithm-centric approaches such as the Demeter method (Lieberherr, 1996) which attempts to achieve adaptivity by decoupling the details of a data structure from the operations on those structures. In the Demeter method, executable programs are customised from high-level 'adaptive' programs. These generic programs are linked to detailed data structures via 'class dictionary graphs'.

A further classification that needs to be made is the time that adaptation occurs during the development process of the system. As pointed out above, a distinction is often made between *adaptable* software that is easily modifiable at design time, and *adaptive* software that can modify itself at run-time (Herring, 2002). In software terms, this distinction between adaptable and adaptive is rather crude. McKinley, Sadjadi et al. (2004) provide a more nuanced classification. Composition, a type of adaptive operation, can occur at development, compile, link, load and run times. The degree of adaptability in software systems is classified as hardwired, customisable, configurable, tunable and mutable, depending on the *time* when composition occurs, and whether the

[10] Fogel (1995 pp14-15) points out that "genetically hard-coded behaviour is inherently brittle", that is unable to adapt to *unexpected* perturbations in the environment. He cites the example of hunting wasps inability to adapt to the smallest changes in the environment that change the preconditions for a behaviour.

composition is *static* or *dynamic*. In this work we will refer to software systems as *adaptive* if they can change themselves at runtime without recompilation. In particular, we are interested in software architectures that implement some form of managed control as suggested by (Shaw, 1995) and have the ability to reconfigure themselves. There have been a number of attempts to develop such adaptive architectures. The next chapter reviews these approaches.

The final distinction we make in our taxonomy is between runtime adaptive architectures that are based on role-based organisational concepts, as discussed in the previous sections of this chapter, and those architectures in which components are the structural nodes. This is what distinguishes our approach from other architectures surveyed in the next chapter.

To conclude, we can characterise the focus of this thesis in terms of the questions on the nature of adaptation which we posed at the beginning of this chapter. Thus, we are concerned with ontogenically adaptive systems that can adapt at runtime to changes in their environment and to changes in their goals and requirements. These systems can adapt to changes by dynamically altering their structure, and by regulating the behaviour over that structure. In system-theoretic terms, we present a framework for building ontogenically adaptive systems with the following properties:

- A flexible structure based on the radical separation of role from role players. This provides two levels of indirection*: indirection of association* between roles, and *indirection of instantiation* between roles and players.
- The dynamic management of these indirections *within* a composite through configuration and regulation. This management is based on the separation of control from process, and requires the runtime monitoring of the system.
- The distribution of management control down through the system. The system is recursively composed self-managing composites that have well-defined boundaries. The processes *between* composites are regulated by a *management system.* This management system is distinct from the *functional system*, and it is these management processes that maintain the organisation of the system.

3

Adaptive Software Architectures

The previous chapter described the characteristics of ontogenically adaptive systems, and placed ontogenic adaptation in the context of adaptation in general. Ontogenically adaptive systems maintain their organisational viability by dynamically altering their structure (configuration), changing their components, and regulating the behaviour over that structure. To recap our formulation:

Ontogenic Adaptation = Structural Plasticity + Component Interchange + Organisational Regulation

In this chapter's literature review, we use the concepts presented in the previous chapter to analyse and classify adaptive software architecture frameworks. *Software architecture* is taken to mean a high-level view of a software system seen as a configuration of components[1] and connectors (Garlan and Shaw, 1993). Connectors between the ports or interfaces of components define the permissible behaviours that can occur between components. In the context of software architecture, our definition of ontogenic adaptation above can be rephrased as

Ontogenically adaptive architecture = Dynamic structure + Management

where management activities monitor the system, reconfigure the structure and regulate the behaviour over that structure.

[1] The word *component* is used very generally here to mean any encapsulated software entity that has behaviour described by an interface. Components in this sense could include objects, components (as per (Szyperski, 1997)), agents, services and so on. While some Architectural Description Languages (ADLs) such as Darwin (Magee and Kramer, 1996) treat connectors as components, here components are taken to be the nodes being connected.

In this chapter we will briefly examine the literature on *dynamic* architectures that can replace their components (component interchange or *indirection of instantiation*) and change the relationships between those components (structural plasticity or *indirection of association*). Having a representation of a flexible, dynamically configurable structure is a necessary, but not a sufficient, condition for an adaptive architectural system. Any adaptive architecture must show how such indirection of its flexible structure is managed. In 1995 Shaw proposed that, in some types of application, an architectural idiom based on control theory is appropriate (Shaw, 1995). Since then many approaches have been proposed that develop ways to control or manage components. While not all management approaches necessarily incorporate control-theoretic notions such as control variables, feedback control, etc., they do separate a controller or manager from the process being controlled. We will examine in more detail architectural *frameworks* that provide ways to *manage* indirection in the structure. Such architectural frameworks describe generalised configurations of types of component that perform systemic functions at a management level of abstraction

Having a dynamic architecture with some management capability does *not*, of course, guarantee that the software system itself will be ontogenically viable, i.e. that it will be able to survive and continue to meet its goals by maintaining its organisational integrity. An effective management regime is also needed. In the literature review that follows later in this chapter, we will survey a range of recent work on adaptive software architecture. In this range there are many variations in both architectural structure and management regimes.

This chapter is structured as follows. The next section examines some general requirements for ontogenic adaptation in software. Section 3.2 distinguishes between two broad approaches to developing adaptive software, namely, those approaches that focus on the structure of the architecture, and those approaches concerned with measuring the quality-of-service (QoS) performance[2]. The third section identifies a number of distinguishing characteristics in the adaptive architectural frameworks we review. Section 3.4 discusses and justifies the scope of the literature review, and briefly discusses the many related areas that are not covered. We then review a number of these frameworks, categorising them according to the distinguishing characteristics listed in the preceding section. We conclude the chapter by presenting a comparative table that summarises the characteristics of the reviewed frameworks.

[2] We use the word *performance* in this thesis in a very general sense, i.e. the actual level of fulfilment of any 'non-functional' requirement.

3.1. Requirements for ontogenic adaptation in software architectures

In the domain of software systems, much research has been aimed at creating systems that can restructure themselves at an architectural level of abstraction – i.e. restructure the relatively course-grained components and connectors that constitute the composite system. In ontogenic adaptation three types of change can be identified. *Functional reconfiguration* we define as the alteration of the *types* of functional relationship between entities in the system. *Non-functional reconfiguration* involves restructuring, but no new types of functional relationship are defined. *Regulation* of a system changes the characteristics of existing relationships in the system so it can maintain its viability and continue to meet its goals. We can distinguish different operations involved in these types of change as applied to software architecture. *Functional reconfiguration* involves one or more of the following operations:

- Addition of a new type of component that has a new functional relationship or interface[3] with the system.
- Removal of a component so that there are no longer components of that type and association in the composite.
- A change in the types of relationship between components

Non-functional reconfiguration involves one or more of the following restructuring operations that change the instances of components of a given type, or that effect the multiplicity of relationships between types:

- Replacement of an instance of a component with another instance that fulfils the same function (implements the same interface).
- Replacement of an instance of a connector with another instance that connects components on the same interfaces (ports).
- Addition of a duplicate component (and associated connectors) that conforms to an existing type functional relationship in the composite
- Removal of a duplicate component, provided there remain component(s) that share the same relationship type. In other words the components involved duplicate the same interface and perform the same function with respect to the composite (for example in parallel processing).

Regulation that does not change the type of function involves:

[3] Interface is used here in the sense of required and provided interfaces of a component that define the functional relationships. The concept of an interface can be extended to a “rich” interface that defines the non-functional aspects of a relationship.

- Altering the parameter(s) of existing components thus regulating the function of components.
- Altering the parameter(s) of existing relationships thus regulating the interactions / connections between components.

As well as a flexible and dynamic structure that permits the above operations these operations need to be controlled and managed. To achieve effective adaptation, the manager of the system or composite must have the following properties:

- A representation of its composite's structure. If a manager/controller of the system is to reconfigure it, the manager must hold a representation of its structure, and be able to sense the actual behaviour over that structure. In a self-adaptive system, this manager is part of the system. The system may also need the ability to reason about the representation of the structure to determine if structure is well-formed, and if it will produce the desired behaviour.
- A means for reconfiguring and regulating the structure. The manager must have functional and non-functional operations, as describe above, for manipulating the structure including the ability to create connections and bind entities to the system. The manager needs to decide how to fix (determine) any indirection created by the flexible structure. (Given its ability to create structure, the manager may also have the ability to create a system from static descriptions.)
- When a change in the system, requirements or environment results in the system no longer meeting its goals, the system needs to know that reconfiguration or regulation is necessary.
- Once the need for reconfiguration is established, the system needs to know what alternative configuration(s) can better meet those goals, and how to safely transition the system to the better configuration. There may also need to be higher levels of control regulation to ensure against unwanted instability resulting from the change process.

3.2. Approaches to representing adaptive architectures

Research efforts in adaptive software architecture address one or more of the above properties of reconfigurable systems. Before reviewing the current literature on adaptive architectural frameworks later in this chapter, we will first distinguish two complementary categories of approach to defining dynamic architectures that allow the structure and behaviour of the system to be reasoned about[4]. As illustrated in Figure 3-1

[4] There are many ways the dynamisms of a software architecture can be classified. In his survey Bradbury (Bradbury, 2004) lists eleven definitions of dynamic change that have been widely cited.

below, the first category is primarily concerned with structure, and making sure that the dynamic structure is well-formed. Structure is the concern of configuration management activities. The second category concentrates on the measurement of the qualities of interactions over a structure. Controlling the quality of interactions is the concern of regulatory management activities. These categories are *not* mutually exclusive but, rather, represent different dominant themes for describing frameworks. As shown in Figure 3-1, these modes of description can be further classified according to the mechanisms for achieving configuration and/or regulation. Some of the frameworks we discuss, such as contract-oriented frameworks, can be seen as a mix of structure and quality-centric descriptions. We discuss these categories in more detail below.

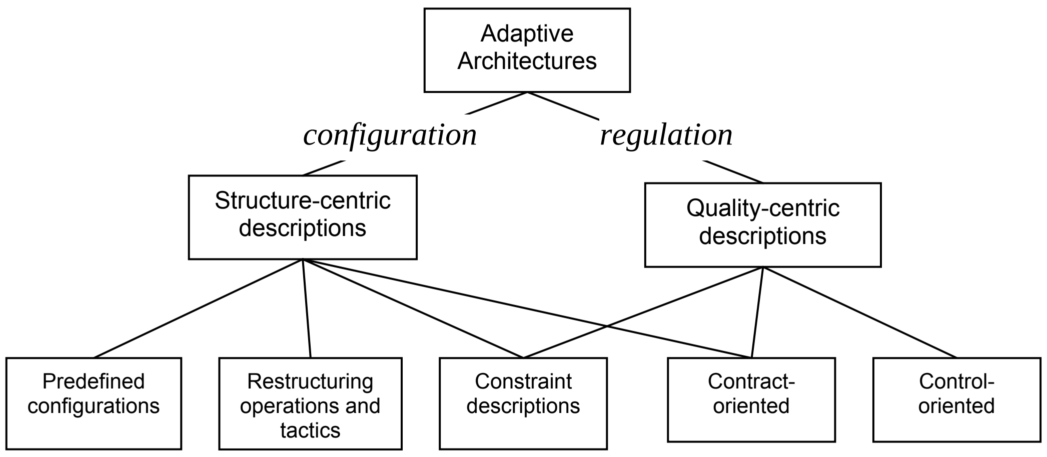

Adaptive Architectures
configuration
regulation
Structure-centric descriptions
Quality-centric descriptions
Predefined configurations
Restructuring operations and tactics
Constraint descriptions
Contract-oriented
Control-oriented

Figure 3-1: Representing dynamic architectures

3.2.1. Structure-centric descriptions

A number of Architecture Description Languages (ADLs), such as Dynamic Wright (Allen, Douence and Garlan, 1998) and Darwin (Magee and Kramer, 1996), attempt to represent architectures with a dynamic structure. Approaches that are primarily concerned with functional change are largely formal. They are motivated both by the need to compose functionally well-formed systems, and the need to express the dynamic transformation of the structure. Bradbury (2004) surveys a number of formal dynamic architecture languages and evaluates the extent to which the formalisms support the specification of self-managing systems. The formalisms include graph-based, logic, process algebra and other approaches. Each approach has various strengths. Bradbury examines the relative expressiveness of the languages in terms of component and connector addition/removal to/from the structure. Formalisms also vary according to whether they emphasise the behaviour of the system (as, say, naturally expressed in a process algebra) as opposed to an emphasis on the structure of the components and connectors (as, say, naturally expressed in a graph grammar).

Dynamic architectures can also be differentiated according to whether the description is positive (describes what is) or negative (describes what cannot be). Positive descriptions *enumerate* the acceptable configuration states of a structure. Such approaches have less ability to adapt to a wide range of environmental variation than architectures where possible structural configurations are defined using constraints. Enumerated configurations have to be thought of by the designer in advance. On the other hand, constraint-based approaches potentially provide more indirection in the structure because they only define what is *not* a valid configuration. However, there is a cost to this greater indirection: the runtime mechanisms for the selection of components and for their configuration need to be more capable, and thus require powerful formalisms for reasoning about the structure. In order of increasingly greater indirection in the description, dynamic architectures can be classified as follows:

- Types of components and connectors are fully specified prior to runtime, as are the alternative configurations (as, for example in Dynamic Wright (Allen, Douence and Garlan, 1998)). Selection of configuration is predetermined in that particular configurations are predicated on particular environmental states.
- Types of components and connectors are specified prior to runtime, but the selection of components and their association with connectors is determined by predefined rules for manipulating the structure. It is these *operations* on the structure that are selected when reconfiguration is needed, rather than pre-defined configuration states. Operations are often selected by using predefined sequences of operations called *tactics*. e.g. (Garlan, Cheng, Huang et al., 2004)
- New types of components and connectors can be added at runtime, but the selection and combination of components is determined by constraints. At runtime, configurations need to be validated, given the constraints and the available components. Components do not have to be of the same type or even form an equivalence class. For example, the Darwin architecture language (Magee and Kramer, 1996) is designed to dynamically define a set of valid configuration actions given a set of architectural constraints. The constraints are similar to an architectural style (Shaw and Garlan, 1996).

We will not examine the diverse range of dynamic architecture languages here, as this is provided elsewhere (Bradbury, Cordy, Dingel et al., 2004), other than to note that a number of them have been used as the representational basis for adaptive frameworks that are discussed in the review of literature below.

3.2.2. Quality-centric descriptions

Non-functional transformation is the focus of other approaches to describing software architecture[5]. These approaches are concerned with representing structures in which changing performance and other qualities (reliability, resource allocation, security etc) can be represented and managed. If an architecture, typically in the form of a framework, is to implement non-functional adaptation (reconfiguration and regulation) it needs to describe how it performs some, if not all, of the following functions:

- There must be a way of characterising and quantifying non-functional parameter(s) of interest (e.g. latency, reliability, cost). These parameters must be represented both as a *required* value or set of values defined by constraints, and as an *actual* value.
- It needs to provide, or be able to receive inputs from, mechanisms for monitoring the quality attributes of interest.
- Because non-functional properties typically cross-cut the functional structure, those properties may need to be modelled across the system, and across different levels of abstraction. For example, if a system is to have a level of reliability then its components must also have some level of reliability.
- Because a system may have a number of competing qualities it needs to satisfy, there may need to be a way of balancing these requirements to produce an optimal outcome (e.g. throughput versus cost).
- There needs to be some mechanisms and strategies for mitigating ‘under-performance’; that is, regulating or restructuring the system when an actual quality parameter does not fulfil the requirements for that parameter.

While each of the frameworks discussed below possess some of the above properties, none comprehensively address them all.

As was shown in Figure 3-1, a distinction can be made between *contract-oriented* and *control-oriented* approaches to implementing systems with the above regulatory properties. *Control-oriented* approaches regulate entities by monitoring the changes in the values of control variables, and then setting process variables (as described in the previous chapter). *Contract-oriented* approaches regulate the interactions between entities by defining permissible types of interaction and the performance levels of those interactions. Both approaches define required performance levels, have some sort of monitoring mechanism to ensure the required level of performance is being met, and

[5] Outside the context of adaptive software architectures, various approaches to characterising and modelling quality attributes have been developed (e.g. (de Miguel, 2003) provides a summary of approaches to the specification of QoS models).

take action to correct any underperformance. In control systems this involves the controller changing some property of the controlled entity. In contracted systems the entity needs to autonomously meet the requirement. As contracts can also be used to define relationships between components, they can be viewed as connectors that define the structural relationships in a system. Contracts (or ‘connector types’ as in (Allen and Garlan, 1997)) can therefore be used to describe both the structure and quality of relationships, as illustrated in Figure 3-1 above.

3.3. Distinguishing characteristics of adaptive frameworks

While software architectural languages provide ways to describe software architectures, architectural *frameworks* provide archetypal arrangements of generic types of component. The functions defined by adaptive architectural *frameworks* can be differentiated in many ways. As we pointed out at the beginning of this chapter, the management of an ontogenically adaptive architecture involves (re)configuration and regulation activities. The characteristics of these activities, along with the characteristics of the management system itself, vary markedly between frameworks. Using these categories of *configuration*, *regulation* and *management*, we define a number of differentiators which we will use to compare the dynamic architectural frameworks surveyed in the next section of this chapter. These distinguishing characteristics can be seen as an elaboration of the desirable properties for adaptive architectures we listed in Chapter 1; i.e. grounded, exogenous, self-managed, recursive and practical.

1. Configuration

1.1. *Reconfiguration possible at runtime.*
Can the connections between components in the structure be changed at runtime?

1.2. *Composition based on declarative description possible at runtime.*
Is it possible at runtime to create compositions from declarative descriptions, or does the basic structure have to be defined at compile time?

1.3. *Functionally recursive structure.*
Do configured composites of components themselves form a unity that can be configured into larger composites? Can the management regime be scaled to handle the different granularities of such recursive composition?

1.4. *Non-functional restructuring supported.*
Can multiple components of the same type be created in parallel to serve a single functional output?

1.5. *Elements can be substituted (indirection of instantiation supported)*

Can one component be substituted for another component at runtime? Can the components be *safely* substituted?

1.6. *Supports heterogeneous components.*

Can the components in the system be based on different software technologies? For example can objects, components, services, and agents be used in the one composite?

1.7. *Structure is entirely defined and controlled by management.*

Are the components ignorant of the structure i.e. do they use "blind communication" (Oreizy, Gorlick, Taylor et al., 1999), or do the entities that form the structure have to have a representation of (some of) the relationships in that structure?

1.8. *Partial instantiation possible.*

Can an application built using the framework continue to function even if not all components are present at any one time in the structure?

1.9. *Formal reasoning about structure possible.*

Can the structure be formally represented so that it can be reasoned about? For example, can proposed compositions be checked for integrity?

2. Regulation

2.1. *Non-functional regulation possible.*

Does the application have the ability to monitor non-functional properties, and adjust its behaviour accordingly?

2.2. *Control dynamics supported.*

Does the application support control-theoretic concepts such as control of hysteresis, negative feedback and so on?

2.3. *Utility can be defined arbitrarily*

Can the control variables used to regulate the system be defined arbitrarily by the application programmer, or are they intrinsic to the system?

2.4. *Utility requirements can be changed dynamically.*

Can the goal (set-point) to be achieved by some interaction, as measured by a utility function(s), be changed at runtime?

2.5. *Type of utility can be changed dynamically.*

Can different types of utility be added to the monitoring mechanisms at runtime, or does the measurement utility need to be defined at design time?

2.6. *Multi-dimensional utility supported.*

Can multiple utility functions be evaluated for the one component or transaction?

3. Management

3.1. *Mechanisms for determining the need for reconfiguration or regulation are defined.*

Are the mechanisms defined by which the application can monitor its performance, so that it can regulate its behaviour? How does the application define and monitor the preconditions for reconfiguration?

3.2. *Management as separate entity.*

Are management functions encapsulated in a separate aspect or runtime entity?

3.3. *Management exogenous versus endogenous (Arbab, 1998).*

Can management control the application without having access to the internal implementation of the components (blackbox or exogenous coordination) or is management endogenous?

3.4. *Management distributed versus centralised.*

To what extent is management distributed down through the structure? Is there one central manager for the whole application or does each sub-composite have its own manager?

3.5. *Management structure not subject to single point of failure.*

Are there critical management nodes whose failure will lead to failure of the whole application? Even if management is distributed there may still be dependencies between management nodes that can result in single point of failure.

3.6. *Separate management structure.*

If there are separable management entities, do these entities have a network separate from the functional structure?

3.7. *Management can find and/or select components (i.e. resolves indirection of instantiation).*

Are runtime mechanisms defined (or referred to) for finding suitable candidate components, and for selecting the best candidate?

3.8. *Management mechanisms can be superimposed* a posterior *on functional components.*

Can an organisational structure be superimposed on components that have not been designed to participate in such a structure? Exogenous management (point 3.3) is a prerequisite this superimposition.

3.9. *Management is updatable.*

Can management entities be dynamically updated with new strategies for managing their composites? Alternatively, are mechanisms defined for a manager to improve its strategies by learning?

3.10. *Management is substitutable.*

Can a less capable manager be replaced with a more capable manager?

3.11. *Supervisory control possible.*

Can management control be overridden by external control (e.g. a human controller) in some circumstances?

3.12. *Costs of reconfiguration can be estimated.*

Are mechanisms defined to enable the manager to recalculate the costs of reconfiguration, and to prevent unwanted oscillations in the system when reconfiguration occurs?

4. Other

4.1. *Implementation is apparent.*

Is there evidence in the literature that the architecture has been implemented and evaluated?

The above characteristics are a compendium of features relevant to ontogenic adaptation found in the various adaptive architectural frameworks discussed below. As such, none of the frameworks discussed meet all of the above criteria. Some of the frameworks partially meet a criterion: for example, Plastik (Batista, Joolia and Coulson, 2005) supports the definition of multi-dimensional utility, but this can only be defined at design time. There are other desirable criteria that none of the frameworks adequately address. In particular, although some frameworks allow the measurement and regulation of quality attributes of particular components, none adequately define mechanisms for formally reasoning about the aggregated performance of a composition (as, for example, proposed in (Khan and Han, 2005)) . Such mechanisms would allow the aggregated system-level performance to be derived from the individual *actual* performances of the system's components, or the *required* component performances to be derived from a system-level requirement. However, some of the frameworks do provide a basis for the future development of such capability.

3.4. Existing surveys and selection of literature

Surveys have been conducted on dynamic Architectural Description Languages (ADLs). Medvidovic and Taylor (2000), in their survey of ADLs, use dynamism as one of their criteria. A more specific focus on dynamic ADLs can be found in (Bradbury, Cordy, Dingel et al., 2004) (Bradbury, 2004) and a literature review in Georgiadis's thesis (2002). In terms of dynamic architectural *frameworks*, however, much of the work is recent and currently evolving. We are not aware of any comprehensive survey of the adaptive frameworks that takes a broadly architectural perspective. Some of the work on adaptive frameworks has direct antecedents in research into dynamic ADLs, but other adaptive approaches arise from work done on distributed, pervasive, reliable, survivable, autonomous, adaptive middleware, resource-aware real time, control-oriented, grid, service-oriented software systems (to name a few strands).

The following survey of literature on adaptive frameworks has been limited to those approaches that involve some form of structural reconfiguration, and have architectural elements that perform a defined management function. Based on the work's *predominant focus*, we have divided this literature into the categories shown in Figure 3-1 above:

- Structure-centric frameworks
- Control-oriented frameworks
- Contract-oriented frameworks

What this survey does not cover

We have limited this survey to *recently* developed frameworks. Adaptive frameworks developed in the 1990's such as (Kokar, Baclawski and Eracar, 1999; Wermelinger, 1998; Oreizy, Gorlick, Taylor et al., 1999) are not discussed as little subsequent work seems to have eventuated from these early approaches. Nor do we discuss what could be termed *adaptive architectural styles* such as C2 (Medvidovic, Oreizy, Robbins et al., 1996) and Weaves (Gorlick and Razouk, 1991). These styles impose restrictions on how components can be connected, and how they can asynchronously communicate via connectors. Although, in both C2 and Weaves, structures can be arbitrarily complex and can be dynamically manipulated, they do not define any management infrastructure.

We also limit this survey to architectures that can define application specific compositions. Other work has focused on defining generic adaptive middleware. Agha (2002) argues that as software applications becomes more open and mobile, middleware needs to move from being a glue that binds distributed systems together, to being an enabler of dynamic interactions between autonomous actors. McKinley (2004) sees middleware as the logical place to put adaptive behaviour that is related to cross-cutting concerns such as QoS, fault tolerance and security policy. Adaptive frameworks that address or rely heavily on middleware include COCA (Zhou and McKinley, 2005) and CASA (Mukhija and Glinz, 2003). Other approaches see adaptive middleware as the key to ubiquitous computing, with middleware providing adaptive management to distributed applications (Hallsteinsen, Floch and Stav, 2005). Such applications may make use of middleware as enabling technology that can provide services such as introspection and interception. Yet other approaches such as (Hillman and Warren, 2004) focus on providing algorithms and frameworks to ensure that system integrity (synchronisation and state) is maintained during developer-initiated reconfiguration, in particular, during component interchange. The focus in this thesis, however, is on adaptive applications.

Coordination languages also have much in common with dynamic architectures (Cuesta, de la Fuente and Barrio-Solárzano, 2001). In particular, control-oriented coordination languages (Arbab, 1998) compose and control the computational entities in loosely coupled systems. Coordination frameworks by Andrade, Wermelinger and colleagues (Wermelinger, Fiadeiro, Andrade et al., 2001) have much in common with the dynamic architectural frameworks examined here. For example (Wermelinger, Fiadeiro, Andrade et al., 2001; Andrade, Fiadeiro, Gouveia et al., 2002) define a layered architecture with computation, coordination and configuration layers('3C'), and where contracts are first-class entities. However, 3C is not architectural in that its contracts are method-centric rather than entity-centric; that is, they define generic interaction sequences that might involve many parties. Such approaches are primarily focused on synchronisation rather than adaptivity.

Other works that are not included in this survey are those approaches that focus solely on adaptation to variable computational and/or network resources. A number of adaptive control-based architectures focus on reconfiguration as a way of making systems more survivable or dependable. For example, the Willow Survivability Architecture (2001) (Knight, Heimbigner, Wolf et al., 2002) is focused on the reconfiguration of large scale, heterogeneous, distributed systems to achieve network fault tolerance. "The Willow concept derives from a realisation that software configuration control and network fault tolerance are two different aspects of the general problem of overall control of distributed systems" (ibid). While there are many conceptual similarities between Willow and the architectures discussed below, in Willow the emphasis is on the sensing and maintenance of wide-area network state. Similarly, work by de Lemos and colleagues (de Lemos and Fiadeiro, 2002) propose architectures that are fault tolerant. Likewise, the SMART (State Model Adaptive Run Time) framework (Cangussu, Cooper and Li, 2004) focuses on applying control theory in the form of linear state feedback models of the computational environment. Resources (CPU, memory, bandwidth, etc) are monitored. A model of the 'dominant behaviour', given these constraints, is formed, and then used as the basis for selecting off-the-shelf components from a repository. This approach is not architectural in the sense that the *structural relationships* in the system do not change. The focus is only on component replacement.

Another control-centric approach is IBM's *autonomic computing* initiative (Ganek and Corbi, 2003). Like the control-based frameworks discussed below, autonomic computing conceives of a control-loop with three phases: sense-evaluate-act. As originally envisioned, autonomic computing covers four aspects of self-management: self-configuration; self-optimisation, self-healing; and self-protection (Kephart and Chess, 2003). This broad conceptual vision has much in common with the themes of this thesis. However, rather than being a particular framework or methodology, autonomic computing is an umbrella that covers a diverse range of techniques, tools and infrastructure platforms[6].

Current work on Web Service frameworks and standards also addresses many of the same issues as dynamic architectures including dynamic composition, service selection and management. Such parallels are unsurprising as the issues that confront adaptive systems at a general architectural level still need to be solved in technologically-specific approaches, such as Web services. Like dynamic software architectures, work on Web services needs to address both functional and non-functional adaptation. Service composition standards such as BPEL4WS (BEA Systems, IBM, Microsoft et al., 2003) are not adaptive per se. Consequently there has been much recent focus on making service composition more flexible. A recent overview of dynamic workflow-based composition can be found in Zirpins, Lamersdorf et al. (2004). These approaches focus on adaptive processes using process abstraction, rather than focussing on adaptive structures as in architectural approaches. In terms of non-functional adaptation, monitoring of services has also been addressed. Web Services Distributed Management of Web Services WSDM-MOWS defines "manageability" interfaces for Web services (OASIS, 2005) which could provide a starting point for building adaptive Web services. Li, Han et al. (2005) show how the interactive behaviour of a service can be declared and monitored to see whether its behaviour conforms with the composition's requirements. Baresi, Ghezzi, et al. (2004) use external 'Smart Monitors' services rather than application-based monitors. However, there is no mechanism for 'monitoring the monitors' (organisers and contracts) as in a recursive structure. Ludwig, Dan, et al. (2004) propose the Cremona framework that addresses many of the same issues as this thesis but focuses on external service level agreements (SLAs) based on WS-Agreement (Global Grid Forum, 2004)

[6] In practice, IBM now brands "autonomic" any IT management software that conforms to IBM's standards.

rather than on the management of adaptation. Likewise, much work has be done on dynamic SLAs in the context of Service-Oriented Computing paradigm, e.g. (IBM Corporation, 2003; Skene, Lamanna and Emmerich, 2004; Tosic and Pagurek, 2005). These 'external' mechanisms are outside the scope of this review and thesis. However, later in this thesis, we show how the ROAD framework can be applied to Web service composition (see also (Colman and Han, 2006a; Colman, Pham, Han et al., 2006)).

Finally, approaches that are solely agent-oriented are not covered by this survey. Agent-oriented methodologies take as a *sine qua non* open and dynamic environments to which agents have to adapt. A major on-going research issue for multi-agent systems (MAS) is how to achieve system level goals from collections of individual agents. While much of the work to date has focused on negotiated team building and organising mechanisms such as norms, there is a growing recognition of the need for organisational *structures* (e.g. (Jaime Sichman, 2005),(Dignum, 2003)). A number of methodologies developed, such as Gaia (Zambonelli, Jennings and Wooldridge, 2000), explicitly address the need for organisational structure in software. We refer to MAS literature again in Chapter 5 when we come to examine the nature of roles in software, but we do not examine MAS approaches further in this review, as MAS adaptation is largely a result of individual agents' deliberations rather than organisational change.

3.5. Structure-centric frameworks

As pointed out above in Section 3.2.1, structure-centric descriptions of architectures can be loosely classified as those that describe sets of valid configurations, those that define reconfiguration operations and tactics, and those that define constraints. In this subsection we discuss three structure-centric frameworks that use various combinations of these descriptions.

3.5.1. Darwin based frameworks – using constraints

Rather than define what an architecture *is*, constraint-oriented approaches define what an architecture *is not*. As discussed above, such an approach potentially provides greater indirection, and thus adaptivity, to the structure, but at the cost of having to resolve that indirection at runtime. To do this the manager needs to be able to generate and reason about valid alternative structures. Constraint-oriented approaches, therefore, need appropriately powerful ADL formalisms. The constraint-oriented approaches that are examined here do not describe any mechanisms for QoS control. Georgiadis (2002; Georgiadis, Magee and Kramer, 2002) proposes a runtime architecture based on the Darwin architecture language (Magee and Kramer, 1996). The Alloy language

(Jackson, 2002) is used to model Darwin-compliant components[7] so that structural constraints can be expressed and analysed. This approach uses architectural constraints as the basis for the specification, design and implementation of *self-organising* structures for distributed systems. The advantage of having a self-organising architecture is that there is no central configuration manager that can fail. Self-organisation is achieved through each runtime component having a component manager that maintains a representation of the enclosing structure (the configuration view in Figure 3-2 below). The component manager is responsible for the connectors to the *required* ports of other components. If there is any change in these connections, the component broadcasts the change to all other component managers. This enables each of the component's configuration models to be kept consistent.

The manager of each component is responsible for checking that the changes it plans are consistent with the architectural constraints it holds. If the component fails, it only affects the services it *provides,* and the connections to these services are always controlled by the 'client' component (components control their required ports). This way the failed component can always be replaced.

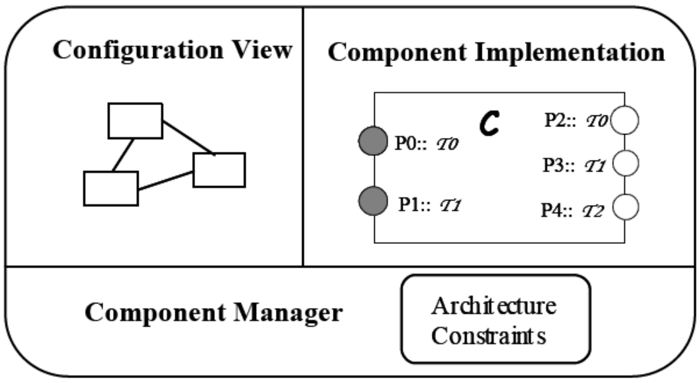


Figure 3-2: Self-managed components (from (Georgiadis, Magee and Kramer, 2002))

This approach has a number of limitations. While Darwin's formalism enables automated reasoning on the structure of the architecture, it only expresses structural relations. There is no provision in the framework for modelling non-functional properties. Nor is there any provision for instrumenting the structure so that performance can be measured.

[7] A component, in Darwin, is a container of provided and required services. Services are provided and required via ports. Ports are typed with the interface that is used to access the service. A component is always associated with the same fixed number of ports during its lifetime. Conversely, ports should always be contained by the same simple component during their lifetime. The set of ports that belong to a component is the union of its provisions and requirements.

The examples used by Georgiadis show how to construct simple architectural styles (such as a pipeline) from constraints. Whether more complex domain-specific architectures can be defined from constraints is problematic. For example, in the realm of agent-oriented programming it has proved difficult to produce purposeful behaviour purely from constraints formulated at design time (Shoham and Tennenholtz, 1995). Waewsawangwong (2004) recognises the difficulty in deciding the architectural structure at runtime, based on constraints alone. He proposes an extension to Georgiadis's work that uses *tactics*. Tactics imperatively specify how a component can assemble or reorganise itself in order to satisfy a given set of constraints. This work appears to be at a very preliminary stage.

While the aim of this approach is to create distributed management, this comes at the cost of requiring global communication between all the components. However, hierarchical composition that can be expressed in Darwin may mitigate this problem to some extent. In a hierarchy the scope of the shared model that has to be maintained by the components would be reduced, thus, in turn, reducing communication overhead and the chance of inconsistencies between the various copies of the model that the components hold. However, hierarchy comes at the cost of reintroducing single-point failure problems that are avoided in fully distributed control.

Other limitations of this approach are as follows. It is not clear how changes, that simultaneous effect a number of components, can be orchestrated. Nor is there an ability to validate structure or behaviour at the system level. This is because checking is at the component level, and system-level behaviour can only be represented indirectly.

Component managers (or the designer of the constraints) presumably need a white-box understanding of the component to derive their constraints. Therefore, coordination is not exogenous, in that it cannot be imposed on blackbox components. Nor is management separable from the components. There is no way to introduce "smarter" management at runtime (while rigorous, the form of structural management described is very limited).

Each component conducts instantiation of its required ports using a selector function (at the time of writing this was manually coded rather than automated). As there is no way of reasoning across the structure, how is it possible to know in advance if there is a valid instantiation of the architecture? The following claim that the "architecture stabilizes when all those required ports that can be bound are bound. Stability is guaranteed in the absence of continuing failure for those systems in which configuration rules guarantee monotonically increasing binding." (Georgiadis, Magee and Kramer, 2002 p36). This assumption, that stability will be reached, seems to assume that compatible components will be found.

3.5.2. Plastik - using pre-defined configuration actions and constraints

Plastik (Batista, Joolia and Coulson, 2005) is a framework that supports a formally specified runtime reconfiguration of systems through the integration of the ACME/Armani ADL (Monroe, 2000) with a reflective component level runtime (OpenCOM (Coulson, Blair, Grace et al., 2004)). The ADL description has two sub-levels, as illustrated in Figure 3-3 below. The *style level* defines generic patterns (e.g. protocol stack style) by setting constraints on the way types of component, connectors and interface operations ('properties') can be composed. A configuration defined by a style is encapsulated in a 'component framework'. The *instance level* particularises the style for a specific context (e.g. TCP/IP stack).

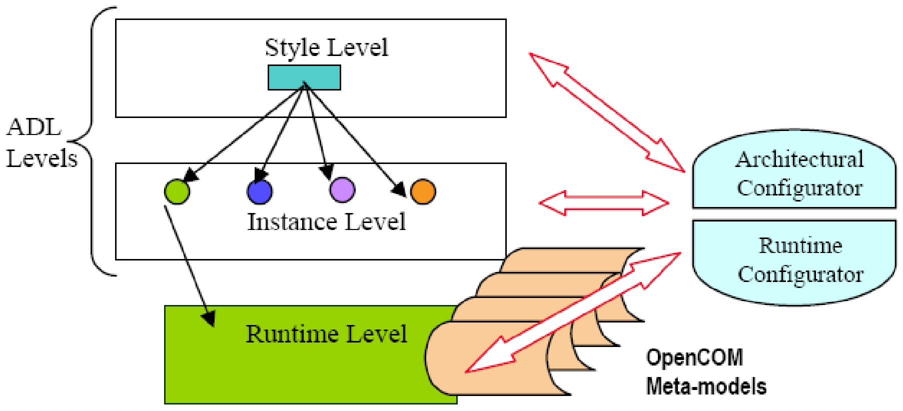


Figure 3-3: Plastik's System Architecture (from (Batista, Joolia and Coulson, 2005))

The 'system configurator' is also divided into two levels. The singleton architectural configurator is responsible for accepting and validating reconfiguration requests from the ADL levels, while each deployed 'component framework' has a runtime configurator that manages the runtime level. Constraints at the ADL level are compiled into finite state machines in the runtime configurators. These configurators are implemented in a scripting language that is generated by a compiler. This script instantiates OpenCOM elements that correspond to ADL-level specifications.

Plastik supports 'programmed' and 'ad-hoc' reconfiguration. Programmed reconfigurations can be foreseen at design time and are expressed as 'predicate-action' specifications. Ad-hoc reconfiguration, on the other hand, specifies certain invariants which configurations cannot violate. Ad-hoc configurations are not specified at the ADL level (only the constraints are specified), but change can be initiated from either the ADL and runtime levels. Change from the ADL level can be initiated by submitting an architectural modification script to the architectural configurator, which is then compiled into 'diff' script for the runtime configurator.

The key advantage to Plastik's approach is that it combines high-level reconfiguration concepts with a robust runtime component framework. However, the separation of architectural and runtime layers into two separate representations connected by compiled scripts, creates the problem of keeping the two representations synchronised. This is particularly so because change can be initiated at either level. There is also no discussion, in the work reviewed, of the nature of the runtime conditions that can trigger reconfiguration. Nor is there a way to explicitly model non-functional requirement or change. The only reconfiguration operations are the addition and removal of components. Monitoring is not part of the framework but is something that, it is claimed, can be provided by the components or a third party. At the time (Batista, Joolia and Coulson, 2005) was written, the Plastik system had not been fully implemented, although "key aspects of the design" had been trialled.

3.5.3. ArchJava – creating predefined configurations

ArchJava (Aldrich, Chambers and Notkin, 2002) is an extension of Java that allows the structural architectural description to be written in implementation code. The authors see the advantages to not having a separate ADL as promoting traceability and ensuring consistency between the architecture and the fine grained implementation. The ArchJava language adds architectural constructs to support *components*, *connections* and *ports*. A component is a special type of object that communicates with other objects in a structured way via ports. Regular method calls are not allowed. Ports represent logical communication channels that can be specified as *provides*, *requires*, and *broadcasts*. Components can be composed of other connected components. Nested subcomponents can either be statically or dynamically created. Dynamic components can be created with a parent component using a `new` operator, as is used to create ordinary Java objects. Connectors can also be dynamically created, and will be removed when they are no longer referenced.

While ArchJava does not address the management aspects of adaptive systems, an extension has been proposed that uses custom connectors (Aldrich, Sazawal, Chambers et al., 2002) based on ArchJava. However the proposed framework only describes the structural aspects of an application, and does not address behavioural or non-functional aspects. It also assumes synchronous communication between components. Despite this, ArchJava potentially provides a starting point for a generic approach to building adaptive applications.

3.6. Control-oriented frameworks – taking non-functional requirements into account

Other frameworks reconfigure and regulate themselves in response to perceived changes in qualities of interest. Control-oriented approaches to architectural adaptation adopt the paradigm suggested by Shaw (1995); that is, of software systems being control systems. These frameworks define a control-loop with three phases: sense-evaluate-act. Software is envisaged with a separate control component(s) or layer that monitors the system and adapts the structure to changing environments or requirements. Much recent work on adaptive architectures has originated from Carnegie Mellon University (Cheng, Garlan and Schmerl, 2005; Huang and Steenkiste, 2004; Garlan, Cheng, Huang et al., 2004; Cheng, Huang, Garlan et al., 2004; Garlan, Poladian, Schmerl et al., 2004). In these related approaches, the system's architecture is used as the control model in the runtime system. This model makes the system's topology and behavioural constraints explicit.

3.6.1. Rainbow

Rainbow (Garlan, Cheng, Huang et al., 2004; Cheng, Garlan and Schmerl, 2005) is a framework that is designed to provide a reusable infrastructure together with mechanisms for specialising the infrastructure to specific systems. The reusable units in the Rainbow framework are in three layers:

- System-layer infrastructure. Provides an interface to the functional system
    - Probes for measuring the system
    - A resource discovery mechanism to find new resources based on some criteria
    - An effector for carrying out system modification
- Architecture-layer infrastructure. Maintains the representation of the architecture and plans and executes adaptations. It includes a
    - Model manager that provides access to the architectural model
    - Constraint evaluator that checks the model periodically and triggers adaptation if constraints are violated
    - Adaptation engine that determines what adaptations (strategies and tactics) need to be performed. These are then executed by the Adaptation engine.

- Translation infrastructure. Maps information across the abstraction gap between the system and architecture layers – for example, translates an architectural level change operation into a system-level operator.

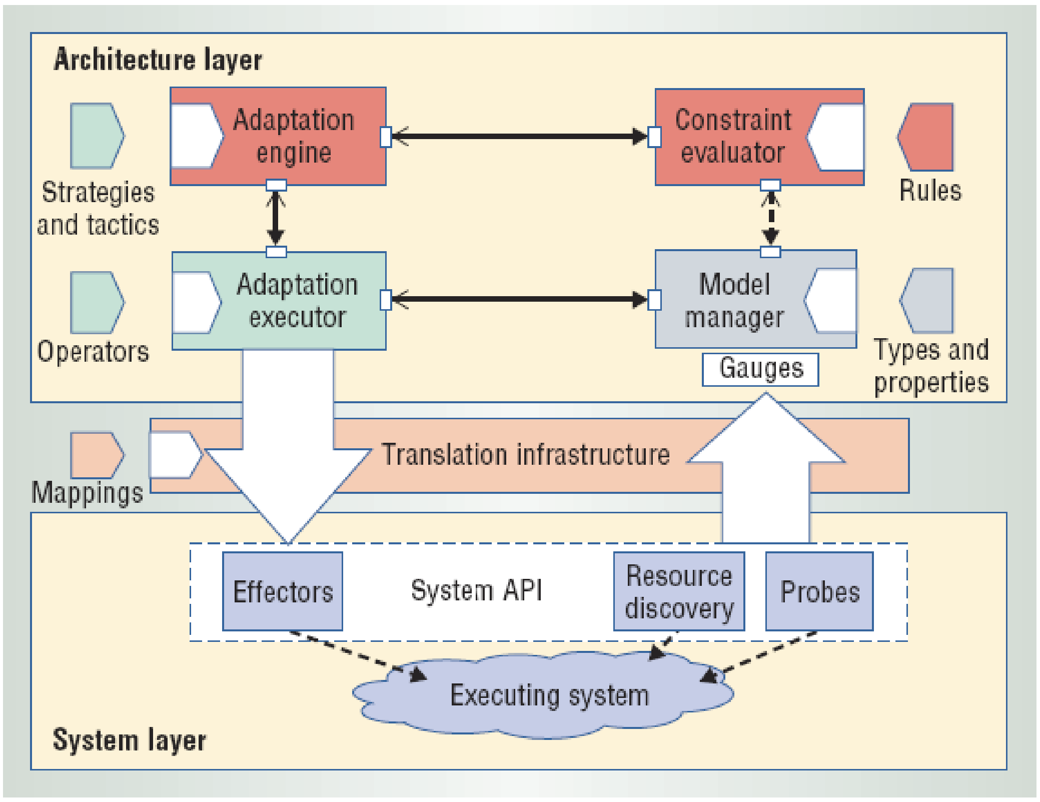


Figure 3-4: Rainbow framework (from (Garlan, Cheng, Huang et al., 2004))

The authors believe that common architectural styles will be able to be extended with "adaptation styles"; that is, with prototypical adaptation operators and strategies. Adaptation operators define reconfiguration operators common to the style – for example *AddService* and *RemoveService*. Adaptation strategies specify the changes to be made to the system in response to the underperformance of some requirement – whether functional or non-functional. Given the constraints and a determination of the problem, a tactic is used to mitigate the problem.

The Rainbow framework assumes that the management framework has access to some measurement, resource discovery and effecting mechanism to observe and change the functional system. In this sense, while Rainbow is *exogenous* to the functional system, the management regime cannot be superimposed *retrospectively* on the functional system. The functional system must have the necessary instrumentation to allow itself to be monitored, and must also have the mechanisms to allow its structure to be reconfigured. Applications, therefore, either need to be designed to comply with the requirements of the management framework (and these may not be known at design-time), or must conform to some common middleware standards. The adaptability of the management framework is therefore limited, because the required probes and effectors may not be present in the functional system or middleware.

In Rainbow, adaptation strategies are globally defined across the architecture. The authors point out that having a central representation of the architecture makes the system subject to single-point failure. Application examples focus on network reconfiguration. They suggest, as future work, applying Rainbow instances to multiple subsystems of a distributed system. These instances would then need to be coordinated. However, it is not clear how generalisable such strategies would be across subsystems.

3.6.2. Self-management modules

A variation of the Rainbow framework is found in (Cheng, Huang, Garlan et al., 2004). There is a recognition that management concerns (composition, change, performance, cost-of-service, etc.) are multi-dimensional and need various utility models. This approach encapsulates these models that cross-cut the functional system in "Self-management modules" (SMs). Each module is responsible for a different management concern across the system. The paper addresses the problem of coordination of multiple SMs to ensure desirable system-level behaviour.

Coordination between SMs is necessary at each phase of the sense-evaluate-act control-loop. As shown in Figure 3-5 below, coordination in each of the phases is, respectively, addressed by:

1. **Sensing**: Consistent system access – so that all SMs are sensing the same system data and translate that data in the same way.
2. **Evaluation**: Non-conflicting decision making – common utility models need to be used to interpret the data on resources. For example, what is "high cost" or unacceptable "slow speed"? Mechanisms for resolving/negotiating conflicting change requirements of SMs are also needed.
3. **Action**: Consistent model – when more that one SM wants to make a change to the system, they must be able to share information to coordinate action, and consequently need a consistent model of the architecture of the system.

The paper addresses the first and third of these coordination aspects. The second, and presumably much more problematic aspect of coordinated decision making, is left to future work – although they do suggest some general control patterns like "master-slave", "democracy"; "balance-of-power" (mutual veto); etc.

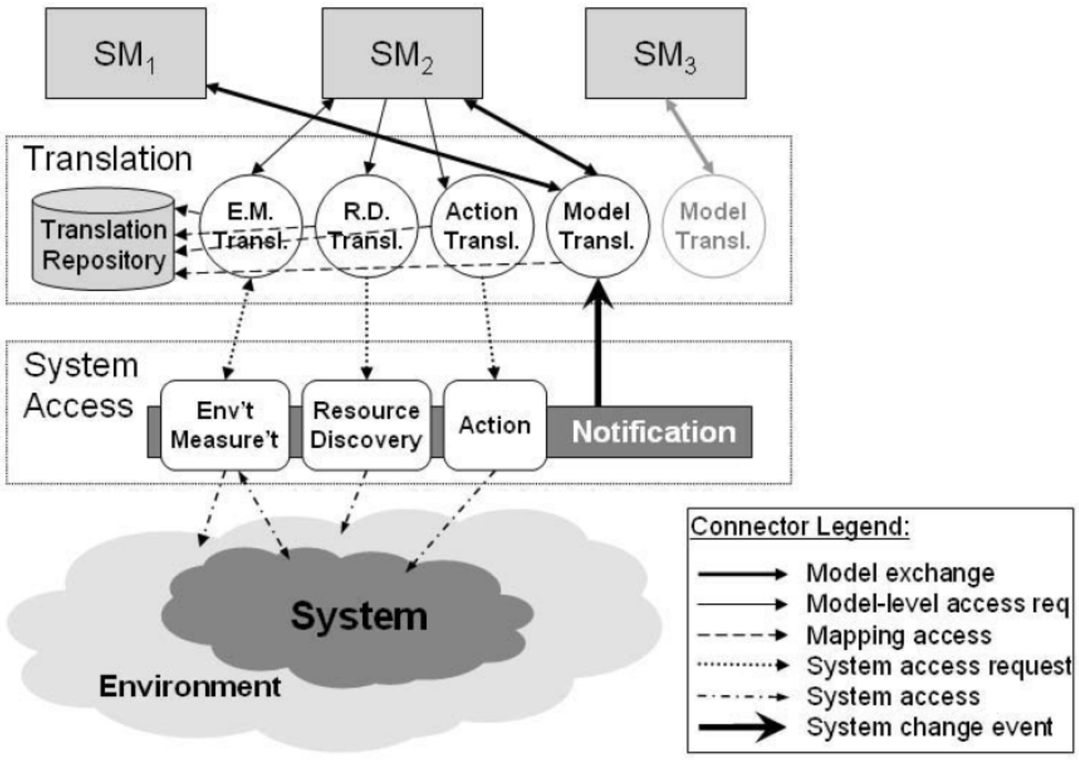


Figure 3-5: Rainbow framework extended with multiple Self-Managed Modules (SMs) (from (Cheng, Huang, Garlan et al., 2004))

In their example, they categorise Libra (a global configuration approach) and Rainbow as SMs, and attempt to coordinate the management between these. Libra's focus is "global configuration", whereas Rainbow follows an "incremental adaptation" approach[8]. These SMs are differentiated as implementations with different scopes of action, rather than being SMs that address various utility functions. (The aim, presumably, is to make use of COTS management components).

This variation of Rainbow has a number of limitations. The term "self-management module" to describe these cross-cutting management modules seems to be a misnomer. The scope of application of these modules is still global, like the original Rainbow framework. The functional components within the framework are not self-managed. Failure of a SM will lead to failure of the management for that concern (e.g. accounting) across the whole system.

While this approach modularises management concerns, these concerns are rarely orthogonal. Typically, non-functional requirements (performance, cost, reliability etc.) are highly inter-related, and need to be traded-off against each other. It is not clear that

[8] An alternative to the functional/non-functional distinction is provided by (Cheng, Huang, Garlan et al., 2004), who distinguish between "global configuration" and "incremental repair". We would argue that a definition based on scope of change is not particularly useful, as the same types of reconfiguration processes occur at all levels of granularity.

the separation of non-orthogonal management concerns into separate modules is a sensible approach. While the separation of these concerns may add analytical clarity and perhaps enhance reuse, the management system must sense, decide and act as a unity. Such an approach creates a problem of synthesis of these concerns. Attempting to apply coordination management at a global level, as this framework does, comes at the cost of considerable additional complexity. (The authors themselves point to the problem of the explosion of coordination paths between SMs). A better approach may be to provide the as-needed ability to add coordination at the interaction level, as is done in contract-based approaches. Conflicts between NFRs (coordination complexities) can then be resolved at a more local level.

For these reasons, we would argue that decomposing a management system on the basis of management concern rather than function, leads to a complex, hard-to-change, non-scalable and ultimately unmanageable organisation. Nor does this form of decomposition address the problem of single-point failure.

3.6.3. Rainbow variant (Huang et al.)

Another framework related to Rainbow is described in (Huang and Steenkiste, 2005; Huang and Steenkiste, 2004). Like Rainbow, it adopts an externalised approach to adaptation, but the focus is on allowing developers to add run-time adaptability to their services. The framework assumes the initial configuration of the service is already completed, and that infrastructure for measurement and service discovery are already in place. The adaptability provided by the framework is limited to self-adaptation capabilities for a single component; that is, changing its parameters or replacing it. The developer defines coordination policies and adaptation strategies that can comply with the framework's knowledge representation, and can, in turn, be used by the generic "synthesizer" in the runtime framework to compose and adapt the service. This approach has similarities to (Georgiadis, 2002) (as described above) in that management is focused on the individual component (service). In this case, however, a knowledge representation of the service's policies and strategies is passed to a generalised adaptation coordinator/manager which is responsible for carrying out the adaptation. In (Georgiadis, 2002), adaptation decision making and action is the responsibility of the component itself. The figure below illustrates Huang et al.'s approach to "self-adaptation"

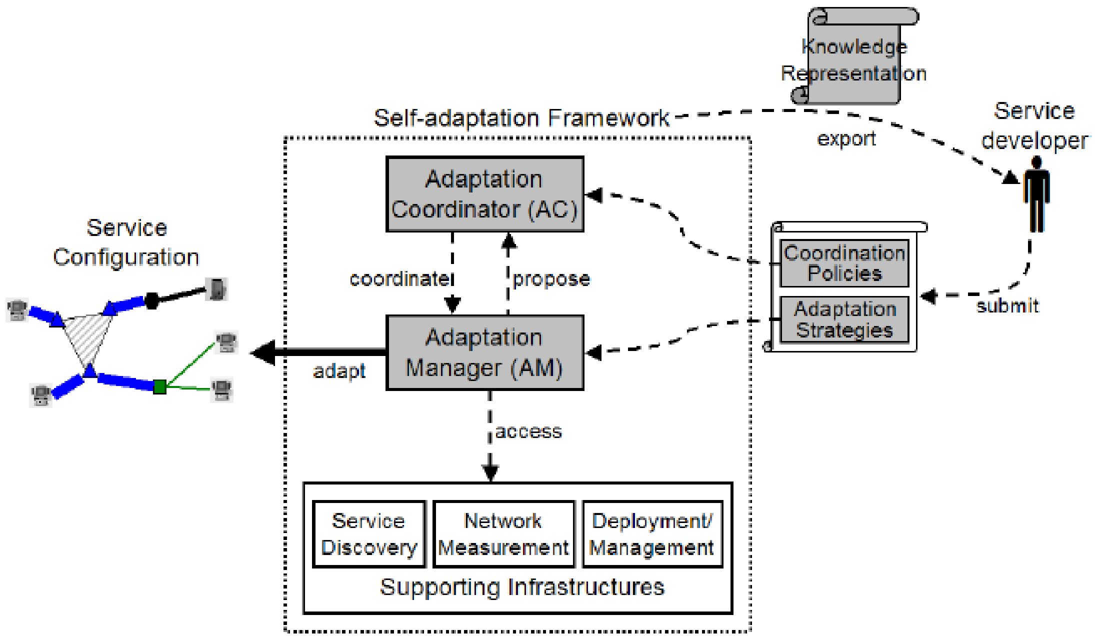


Figure 3-6: Architecture for runtime local adaptation support (from (Huang and Steenkiste, 2005))

Adaptation strategies are the same as in Rainbow – when a *constraint* is violated, the *problem* is determined and a *tactic* (set of actions) employed to mitigate the problem. The actions can include altering component parameters, as well as component addition and/or replacement. Strategies are created by the service developer specifying rules dictating what mitigation actions will be taken when particular events occur. These strategies are specified using the framework's API.

In the framework, the Adaptation Manager (AM) proposes a change of configuration to the Adaptation Coordinator (AC). The AC's task is to coordinate various proposals to resolve conflicts and to identify incompatible strategies.

The authors distinguish their action-event approach from the a "utility function" approach (Walsh, Tesauro, Kephart et al., 2004). However, it would seem to us that a utility function is just a continuous action-event function rather than a discrete function, as described in the paper. Both types of function take, as an input, a change in state of the system or its environment, and then output preconditions for a management action. Continuous utility functions can define events that are triggered when the utility function passes certain discrete threshold values. Such events could in turn trigger, for example, a reconfiguration action.

This variation of the Rainbow framework has the advantage of being able to focus adaptation strategies to specific services/components, and to have those strategies defined externally. This decoupling of (1) the mechanisms for executing a change from (2) the definition of the high-level strategy to carry out the change, is a useful property of adaptive systems. As will be seen later in this thesis, our approach of separating management roles and players facilitates this decoupling. Depending on how the framework is implemented, this decoupling could presumably allow strategies to be developed and modified dynamically at runtime, thus allowing for adaptive learning or supervisory control. Such directions for use, composition and repair (a "service recipe") of the service could be declaratively defined, and append to the service in, say, an XML file to the service.

3.6.4. Aura - task based self-adaptation

As we have pointed out above, the need for adaptation can arise from a change in the environment of the system, or from a change in what is required of the system. One example of requirement change is the situation in which pervasive or ubiquitous systems need to change their behaviour to suit the current user, or the user's context. In the Aura project, Garlan et al. (2004) define an adaptive framework for the construction of "task-aware" systems. A task is a "set of services, together with a set of quality attribute preferences expressed as multidimensional utility functions, possibly conditioned by context conditions."

Tasks models capture and model user goals and intent, and represent quality attributes of the services that perform those tasks. These quality goals can be conflicting. The adaptive system needs to find the optimal balance of qualities to suit the user's goals. To do this it needs to evaluate and consolidate multi-dimensional utility functions. For example, in a video application bandwidth, screen size, frame rate might all have to be balanced given the user's goals and resource constraints.

Aura is an infrastructure with three layers. These layers and their respective functions are:

- Task Management (TM) – determines *what* the user needs from the environment at a specific time and location
    - monitor the user's task, context and preferences
    - map the user's task to needs for services in the environment
    - complex tasks: decomposition, plans, context dependencies
- Environment[9] Management (EM) – determines *how* to best configure the environment to support the user's needs

[9] The *Environment* in Aura refers to the user's environment rather than just the computing system's environment.

- monitor environment capabilities and resources
- keeps track of available service suppliers
- map service needs, and user-level state of tasks to available suppliers
- ongoing optimisation of the utility of the environment relative to the user's task

- Environment – comprises the applications and devices that can be configured to support the user's task
  - monitor relevant resources
  - fine grain management of QoS/resource tradeoffs

This framework can be viewed as consisting of two feedback loops – the TM reacts to changes in, and maintains a model of, user preferences and context. The EM, on the hand, monitors the applications, devices and resources, and maintains a model of the technical environment. The EM also has to take account of the cost of change to the configuration of the system, and ensures oscillation is avoided.

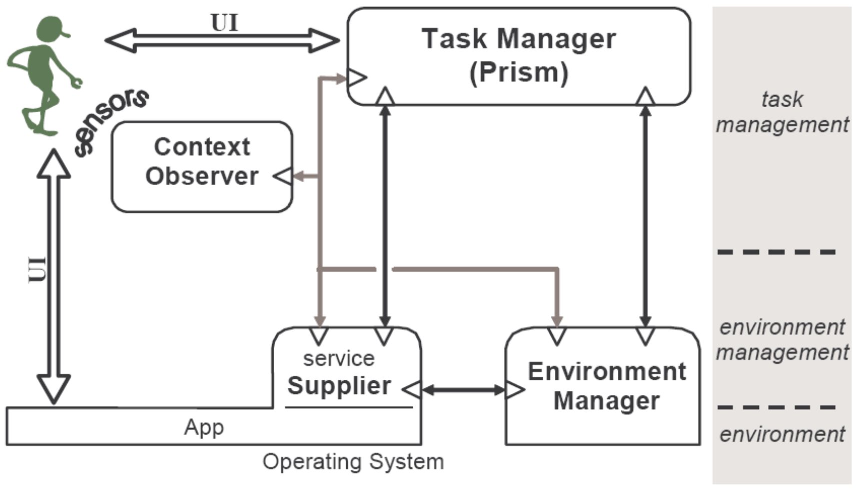


Figure 3-7: The Aura Architecture for Ubiquitous Computing (from (Sousa and Garlan, 2003))

The Aura project introduces an important concept to control-oriented adaptation in software; that is, control loops are needed to adapt to both the external environment (user, system context, problem domain), and the computational/network environment (bandwidth, CPU, memory etc.). The multi-dimensional nature of quality attributes is recognised, and formalisms (albeit crudely modelled) are introduced to calculate optimal utility (Poladian, Sousa, Garlan et al., 2004). This has the potential to complement the extensive work on user modelling, task analysis and adaptive user interfaces (e.g. (Norman, 1984),(Horvitz, 1999),(Taylor, 1988), (Sullivan and Tyler, 1991), (Duce, 1991)). Work on goal-oriented reasoning about non-functional requirements (e.g. (KAOS, 2003), (van Lamsweerde, 2001; van Lamsweerde, 2003) could also be used to extend this approach.

3.6.5. Viable System Architecture

The **Viable System Architecture (VSA)** proposed by (Herring, 2002), takes an explicitly control-theoretic approach to building software systems. This high-level reference architecture defines a set of interfaces for components that accords with the control model postulated in Beer's Viable System Model (Beer, 1984) which we have already discussed in Section 2.2.4 of the previous chapter. Herring takes Beer's concept of a "viable system" and proposes the encapsulation of such systems into "viable components". Each viable component potentially implements all the management subsystems (described in the previous chapter), and has a standard set of interfaces that allows the subsystems within each component to communicate. A viable software system is made up of a recursive hierarchy of viable components. The recursive nature of the architecture, and the relationship between the subsystems, is illustrated in Figure 3-8 below. In the figure, the management systems (2-5) control the Plant system 1. The Plant has three viable subsystems (A, B, C) each with their own controllers (1^A, 1^B, 1^C). There are also typed communication links between the controller in the enclosing component, and the controllers in the subcomponents (not shown).

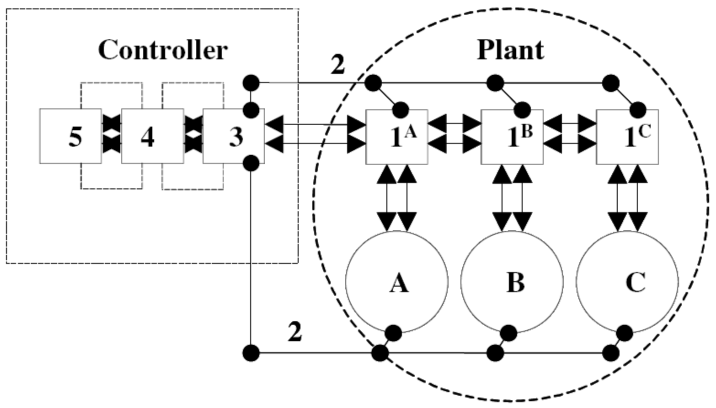


Figure 3-8: Simplified Viable System Model Diagram (from (Herring and Kaplan, 2000))

Viable System Architecture has a number of strengths. The distinction between different types of control provides a more nuanced basis for discussing control in adaptive systems. In particular, it addresses issues such as regulation, stability and homeostasis that are rarely discussed in the context of software architectures. The recursive architecture of self-managed components also provides a way to handle complexity. All components have a common set of abstract management interfaces. However, the downside of this approach is the complexity of the components. In practice, not all components need such a complex structure, and in the examples given in (Herring, 2002) many of the control functions are in fact deprecated. Despite these limitations, VSA does attempt to embody two important principles that help manage complexity in dynamic systems. The first is the strict separation of control/management from process/function. The second is local control – that is, the distribution of control/management down through the structure.

3.7. Contract-oriented frameworks

The *control-oriented frameworks* discussed above focus on the monitoring and control of components through the sensing and manipulation of control variables. *Contract-based frameworks*, on the other hand, exercise control through the (dynamic) specification of the relationships which components must follow. In the frameworks discussed below, there are differences in the types of contract described. Some contracts are *compositional* in that they define the valid configuration(s) of a composite. Other contract-based frameworks view contracts as exercising control by constraining the *interactions* between components. In this sense, contracts are both structure-centric and quality-centric descriptions, as discussed at the beginning of this chapter (Figure 3-1). Contracts can define both the existence of relationships (hence structure), as well as the quality of those relationships. Some frameworks described below, such as ConFract (Collet, Rousseau, Coupaye et al., 2005) and the framework described in this thesis, have contracts that perform both these functions.

As illustrated in Figure 3-9 below, there are two methods of defining and controlling the quality of relationships. The first method is to control the interface of the component involved in any association so that only behaviour acceptable to the contract can occur over that interface. This common approach (as typified by Meyer's (1988) Design-by-Contract) characterises the non-functional properties of the component interface irrespective of its actual relationships.

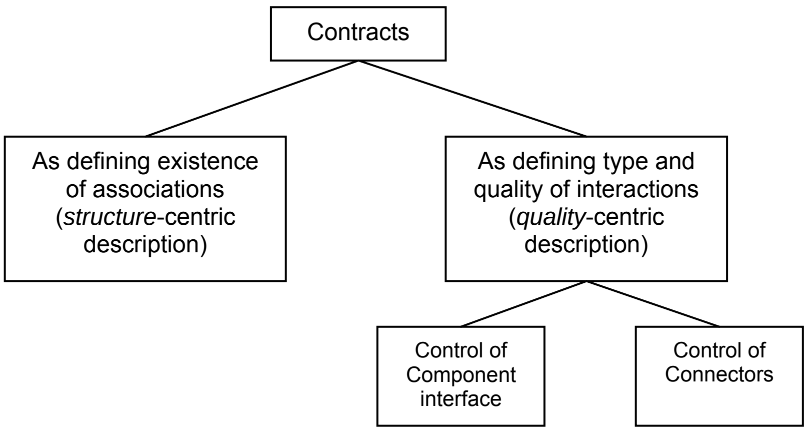

Contracts
As defining existence of associations (*structure*-centric description)
As defining type and quality of interactions (*quality*-centric description)
Control of Component interface
Control of Connectors

Figure 3-9: Aspects of contracts and methods of controlling interactions

The second method focuses on characterising and controlling the *connectors* rather than characterising the components. This is the approach adopted in this thesis. Non-functional relationships can always be reduced to an abstraction over functional relationships. While we tend to think of a non-functional requirement as a property of an entity (role, object, component, etc), it is always a requirement *in relation to* some other entity (or entities). In terms of a contract, a non-functional property of a relationship has both a requirement (obligation) and a state-of-fulfilment of that obligation (performance). Non-functional properties can be viewed as abstractions across functional interactions, even through many such properties (e.g. availability, fault tolerance) may be invariant for all of a component's relationships. While none of the frameworks discussed in this literature review adopt this second method, we highlight this distinction here as a point of contrast to our ROAD framework which is described in the next part of the thesis.

3.7.1. ConFract – contracts for controlling composition and behaviour

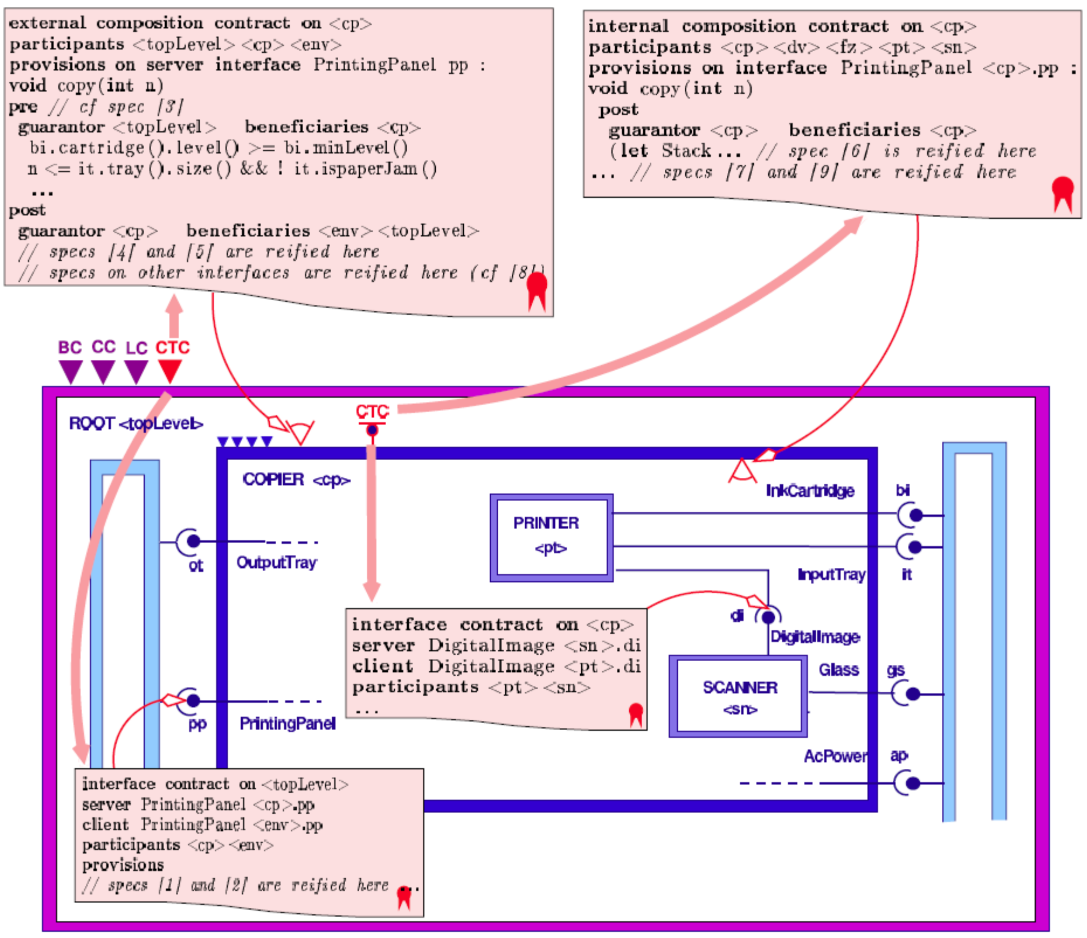
ConFract is a framework that uses contracts to create hierarchical component compositions (Collet, 2001; Chang and Collet, 2005; Collet, Rousseau, Coupaye et al., 2005). It is based on the Fractal component model (Bruneton, Coupaye and Stefani, 2002) which has the following main features:

- Composite components that provide a uniform view of the application at various levels of abstraction
- Shared components to model resources and resource sharing while maintaining component encapsulation
- Reflective capabilities to monitor the running system
- Reconfiguration capabilities to deploy and dynamically reconfigure the system
- Openness, in that almost everything is optional and can be extended.

From an external point of view, Fractal components are connected through server (provided) and client (required) interfaces. A fractal component is formed from a *membrane* and a *content*. The content is composed of other components. The membrane embodies the control behaviour. In particular it can:

- intercept ingoing and outgoing operation invocations
- superimpose a control behaviour on a component's sub-components
- provide an explicit and connected representation of component's sub-components

Every external interface has an associated internal interface. Figure 3-10 below illustrates a Fractal component made up of other Fractal components. The Copier component, contains Printer and Scanner components bound by contracted interfaces. These contracts can be automatically generated from contract specifications, as illustrated in the figure.



```
external composition contract on <cp>
participants <topLevel> <cp> <env>
provisions on server interface PrintingPanel pp :
void copy(int n)
pre // cf spec [3]
 guarantor <topLevel>   beneficiaries <cp>
  bi.cartridge().level() >= bi.minLevel()
  n <= it.tray().size() && ! it.ispaperJam()
  ...
post
 guarantor <cp>   beneficiaries <env> <topLevel>
 // specs [4] and [5] are reified here
 // specs on other interfaces are reified here (cf [8])
```

```
internal composition contract on <cp>
participants <cp> <dv> <fz> <pt> <sn>
provisions on interface PrintingPanel <cp>.pp :
void copy(int n)
 post
  guarantor <cp>   beneficiaries <cp>
  (let Stack... // spec [6] is reified here
... // specs [7] and [9] are reified here
```

```
interface contract on <cp>
server DigitalImage <sn>.di
client DigitalImage <pt>.di
participants <pt> <sn>
...
```

```
interface contract on <topLevel>
server PrintingPanel <cp>.pp
client PrintingPanel <env>.pp
participants <cp> <env>
provisions
// specs [1] and [2] are reified here ...
```

Figure 3-10: Fractal component and contracts (from (Collet, Rousseau, Coupaye et al., 2005))

Collet et al. (2005) point out that interface signatures alone are “insufficient to capture and control the salient properties of an application”. They point to the need to specify “extra-functional”[10] aspects, some of which need to be verified at runtime. ConFract allows behavioural constraints in the form of executable assertions to be specified against interfaces and components. Contracts reify these specification assertions and can be updated dynamically. A contract is “a document negotiated between several parties, the responsibilities of which are clearly established for each provision”. Contracts clearly identify the responsibilities among contract participants so that “developers can precisely handle contract violations”.

[10] We read this as meaning “non-functional”. Indeed, *extra-functional* is probably the better term to express quality aspects because many of these aspects may have functional impacts, but we have chosen to adopt the term *non-functional* in this thesis because of its common usage.

In ConFract, the specifications made up of executable assertions are expressed in CCL-J (based on OCL) and are categorised as *Pre*, *Post*, *Invariant*, *Rely* and *Guarantee*. These assertions extend Meyer's (Meyer, 1988) classic assertions for interface contracts to include state *during* execution. *Rely* is a condition that a method can rely on being true during execution. A method can *Guarantee* that a condition remains true. Each assertion category can consist of zero or more clauses.

As illustrated in Figure 3-10 above, a ConFract system has a number of different types of contract, namely:

- *Interface contracts* are established on the connection point between a pair of client and server interfaces.
- *External composition contracts* located on the external side of each component membrane and express the usage and external behaviour rules of the component
- *Internal composition contracts* are located on the internal side of the composite component membrane, and express the assembly and internal behaviour rules of the implementation of the composite component.

In terms of a contract composition, components are either *guarantors* or *beneficiaries*. If a contract violation occurs, the guarantor (often the contract controller of the enclosing composite) attempts to mitigate the situation by reconfiguring its components. Contracts are managed by contract controllers (CTCs) that are located on the membrane of every component. CTCs react to events in other controllers – the binding, content and life-cycle controllers – to formulate the optimal conditions for its contracts.

In ConFract the scope of a contract can be the whole component. ConFract contracts are thus inherently multi-party, and include both "usage" (interface) and "assembly and implementation" (composition) contracts. Composition contracts explicitly express the composition rules / behaviour of a composite. ConFract external contracts are component-centric. External composition contracts allow behaviour rules to be applied to a component independent of its associations. The responsibility for performance is not part of the ConFract contract itself (contracts only express constraints), but is managed separately by the CTC in the component membrane. In ConFract, the component definition (in terms of it external behaviour as defined by its external composition contracts) is not a separate (role) entity to its internal composition (as expressed by its internal composition contracts). Consequently, replacing a component involves the ConFract system generating (if needed) a new specifications and contracts appropriate to the new component. Indeed, the ConFract system is dynamically built from such contract-based specifications.

In general, the ConFract system is a comprehensive approach to addressing both functional composition and non-functional adaptation. This approach is based on the dynamic creation of various types of contracts from constraint specifications on interfaces and components. However, the complexities of expressing a component composition in terms of constraints may make this approach difficult to apply in practice. In ConFract, components can be changed by opening and closing contracts. Component reconfiguration, as distinct from merely swapping a component, is made possible by changing the specification(s). This specification is then used to generate new contracts at configuration time. Runtime management utilises interception mechanisms provided by the underlying Fractal component platform.

Finally, to characterise ConFract in terms of the schema illustrated in Figure 3-9 of the previous section, ConFract defines both the structure and quality of interactions. However, because interaction control is enforced on the interfaces of the components, the structural and quality aspects must be represented in separate contracts; that is, respectively, the composition and interface contracts.

3.7.2. CASA – configuration selection based on application contracts

Contract-based Adaptive Software Architecture (CASA) (Mukhija and Glinz, 2003; Mukhija and Glinz, 2005b) provides a framework for enabling the development and operation of autonomic applications. As such, the focus of CASA is on adapting to changes in the execution environment, such as computational resources. We examine CASA here because it also addresses changes in the (mobile) user's context; a similar domain to Aura (Sousa and Garlan, 2003). The key features of CASA are:

- Separation of the adaptation concerns of an application from its business concerns
- A runtime system for dealing with the adaptation concerns
- Support for adaptation at various levels of an application
- A contract-based adaptation policy, facilitating changes in the adaptation policy at runtime.

CASA identifies a number of adaptation techniques that can be classified according to the level where the adaptation takes place. These are

- Dynamic change in lower-level services
- Dynamic weaving and unweaving of aspects
- Dynamic recomposition of application components
- Dynamic change in application attributes.

This schema highlights the distinction between the application and its execution environment. It also extends the limited view of architecture; namely, as just configurations of components and connectors to take account of concerns such as security that crosscuts the core functionality (similar to the SMs (Cheng, Huang, Garlan et al., 2004) discussed above).

The adaptation policy for an application is defined by an "application contract". These contracts are external to the application and can be changed at runtime. They define the contexts of interest to the application and a corresponding configuration. Each configuration specifies the resource requirements of the configuration, the components and aspects of the configuration, the callback methods that perform the reconfiguration, and a list of the lower level services related to the configuration.

Every node hosting an autonomic application runs an instance of the CASA Runtime System (CRS) (Mukhija and Glinz, 2005a). As shown in Figure 3-11 below, the CRS monitors the execution environment on behalf of the application, and makes any changes needed in the application. Every time the CRS detects a change in the execution environment (step 1) it evaluates the application contracts of the running applications with respect to the changed state of the execution environment (step 2). The CRS carries out any adaptation needed in the affected applications, in accordance with the adaptation policies specified in the respective application contracts (step 3).

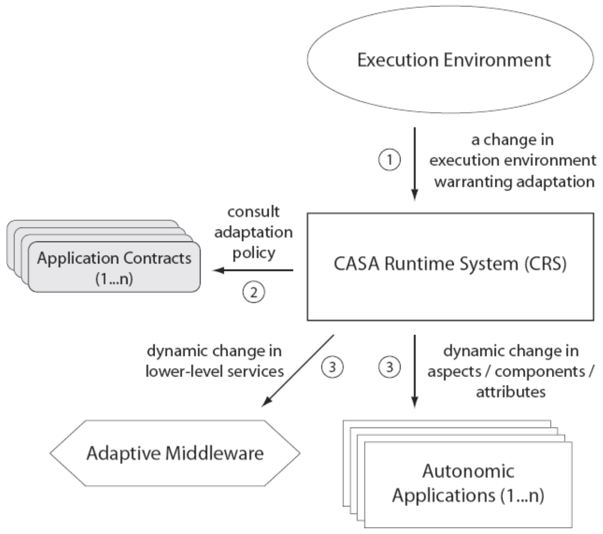

Execution Environment
1
a change in execution environment warranting adaptation
consult adaptation policy
2
Application Contracts (1...n)
CASA Runtime System (CRS)
dynamic change in lower-level services
3
3
dynamic change in aspects / components / attributes
Adaptive Middleware
Autonomic Applications (1...n)

Figure 3-11: Adaptation steps in CASA Framework (from (Mukhija and Glinz, 2005a))

Depending on the current state of the execution environment (contextual information and resources), the appropriate configuration from the application contract is selected and activated by the CRS. Adaptive behaviour in CASA therefore consists of the selection of the first configuration to match the context requirements on a predefined and prioritised list of configurations in the application contract. These contracts can (only) be changed manually by the human user/operator. This provides a form of supervisory control of the system.

CASA does not provide mechanisms for monitoring of resources itself, but relies on the third party middleware. Only computational resources are monitored, rather than application behaviour or domain output. Monitoring of the user context is described at a very general level, and it is not apparent that this has been implemented. Nor is it clear that complex collaborations of a number of applications can be represented or implemented. Although the work on CASA describes a broad vision, implementation at the time of writing appears limited to configuration selection and swapping in a simple prototype.In terms of our schema, illustrated in Figure 3-9, CASA contracts do not define the internal structure of an application, but are rather 'contracts-for-use' that have associated predefined configurations. The interactions controlled by the contract are limited to external interactions between the application and its environment (that may include other CASA applications).

3.8. Summary of framework characteristics

Table 3-1 below summarises the adaptive architectural frameworks reviewed above, according to the characteristic identified in Section 3.3 above. Characteristics marked with a tick ✓ are clearly addressed, or can be clearly inferred, from the work reviewed. Characteristics marked with a cross x are not addressed and it is difficult to see how the framework could support such a feature. Characteristics marked with a tilde ~ are partially supported, or else they are not addressed yet it can be reasonably inferred that they could be supported by the framework.

To be ontogenically adaptive, a framework must provide the capability for structural reconfiguration and regulation of interactions. ArchJava has not been included in the summary because it is a language extension rather than a framework and, as such, does not address management concerns. The Rainbow and ConFract frameworks, in particular, go some way towards meeting these requirements. However, even with these frameworks, much work needs to be done if a comprehensive solution is to be developed. For example, none of the frameworks comprehensively address how the integrity of the system is maintained when components are swapped. When is it safe to swap a component? How is state preserved? How are the costs of reconfiguration evaluated so that decisions can be made on the cost versus benefits of restructuring? How is instability (e.g. feedback oscillations) dampened when restructuring or regulating the system? Some frameworks also lack desirable characteristics such as a clear separation between management and functional entities. Even though a number of frameworks encapsulate these in separate entities, inter-dependencies between management and functional entities mean that the management structure cannot be superimposed on blackbox components. Frameworks that control the quality of interaction do not incorporate formal techniques for ensuring the consistency of the structure.

In Part 2 of the thesis, we introduce the ROAD framework and show how it meets the requirements of ontogenic adaptation. For comparison, the characteristics of our ROAD framework described in this thesis are included in the table. In Part 3, Chapter 11, we evaluate ROAD in detail with respect to the characteristics in the table.

Table 3-1: Summary of the characteristics of adaptive software frameworks

	Georgiadis	**Plastik**	**Rainbow & SMs (Garlan , Cheng)**	**Rainbow (Huang)**	**Aura**	**Viable System Architecture**	**ConFract**	**CASA**	**ROAD**
1. Configuration									
1.1. Reconfiguration possible at runtime.	✓	x	✓	~ [a]	✓	x	✓	~ [b]	✓
1.2. Declarative composition at runtime.	✓	x	x	x	x	x	✓	~	~
1.3. Functionally recursive structure	x	x	~[c]	~	~	✓	✓	x	✓
1.4. Non-functional restructuring supported	~	x	~	~	~	x	✓	✓	✓
1.5. Elements can be substituted	✓	✓	✓	✓	✓	~	✓	✓	✓
1.6. Supports heterogeneous components	x	~	✓	✓	✓	x	x	✓ [d]	✓
1.7. Blind communication	x [e]	x	x	x	x	x	x	x	✓
1.8. Partial instantiation possible	✓	✓	x	x	x	x	✓	x	✓
1.9. Formal composition	✓	✓	~[f]	~	~	x	✓	x	x
2. Regulation		[g]							
2.1. Non-functional regulation possible.	x	~	✓	✓	✓	✓	✓	✓	✓
2.2. Control dynamics supported.	x	x	x	x	~	✓	x	x	~
2.3. Utility can be defined arbitrarily	x	~ [h]	✓	✓	✓	~	✓	x	✓
2.4. Utility requirements changed dynamically	x	x	x	~	✓	~	x	✓	✓
2.5. Type of utility changed dynamically.	x	~	x	~	~	~	~ [i]	✓	✓
2.6. Multi-dimensional utility supported.	x	~	✓	✓	✓	~	✓	✓	✓
3. Management									
3.1. Can determine the need for reconfiguration	x	~	~	~	✓	~	✓	✓	✓
3.2. Management as separate entity.	x	✓	✓	✓	✓	~ [c]	x	✓	✓
3.3. Management exogenous	x	~	✓	✓	✓ [j]	x	x	x	✓
3.4. Management distributed	✓	~ [k]	x	~ [l]	x	✓	✓	✓	✓
3.5. Management structure *not* subject to single point failure	✓	x	x		x	x	x	~	x
3.6. Separate management structure.	~ [m]	✓	~	✓	✓	✓	✓	✓	✓
3.7. Management can find /select entities	x	~ [n]	✓	~	✓	~	~	~ [o]	~
3.8. Management mechanisms superimposed	~ [p]	x	x	x	x	x	x	x	✓
3.9. Management is updatable.	x	✓	~	~	~	x	x	x	~
3.10. Management is substitutable.	x	✓	~	~	~	x	x	x	✓
3.11. Supervisory control possible.	x	x	~	x	✓	✓	~	✓	✓
3.12. Costs of reconfiguration estimable.	x	x	x	x	✓	~	x	x	x
4. Other									
4.1. Implementation is apparent.	✓	~	✓	✓	✓	✓	✓	✓	✓

[a] Only single component replacement or regulation
[b] Contracts can be updated manually at runtime
[c] Claims recursion is possible but not evidenced in discussion or examples
[d] Subject to creating an 'application contract'; external definition of functionality and performance characteristics
[e] Although it might be argued that the component implementations inside management wrappers are blind
[f] The Constraint Evaluator claims to be able to check compositional rules but examples only illustrate QoS
[g] Claims this can be provided by third party
[h] Design time only
[i] Not without redefining and deploying the component
[j] Strategy descriptions are component specific, thus endogenous
[k] Centralised in one ADL description, distributed runtime
[l] Distributed description, centralised action
[m] Separate management communication channel but unstructured broadcast
[n] Claims to support selection but mechanisms are not described
[o] Selection from developer defined list of configurations
[p] Only if components are Darwin compliant

Part II

The ROAD Meta-model

4

Role-Oriented Adaptive Design

In the previous chapter we reviewed architectural frameworks for building runtime adaptive systems. These frameworks facilitate the construction of applications from loosely coupled elements that are then dynamically regulated and reconfigured to meet variable environments or changing goals. This part of the thesis, Part 2, introduces our Role-Oriented Adaptive Design (ROAD) framework and describes the framework at a conceptual level. The description of the implementation of this meta-model is discussed in the following Part 3.

Part 2 is structured as follows. This chapter gives an overview of the basic concepts in ROAD, and introduces an expository example that will be used throughout the subsequent chapters to illustrate the features of the ROAD framework. The rest of the chapters in this part of the thesis then examine each of these concepts in more detail.

4.1. Basic ROAD concepts

The basic elements in the ROAD adaptive framework are roles, players, contracts, organisers and self-managed composites. In the following overview, ROAD's characteristics are described in terms of the features of adaptive architectural frameworks summarised in Table 3-1 of the previous chapter. The specific characteristics listed in that table are referenced below by a number in square brackets [x.y] where x is one of the subcategories 1. Configuration, 2. Regulation, or 3. Management, and y is a label for the adaptive characteristic in the subcategory.

In ROAD, software applications are viewed as organisations — *goal-driven* networks of roles bound together by contracts. There is a radical separation of *roles* from the entities that play those roles. Roles can be played by various *players* (objects, components, services, agents, human operators, etc.) [1.6], in much the same way as a role in a business structure may be played by various employees, departments, or outsourced to external organisations. Similarly, roles in an adaptive ROAD application can be played by players within the organisation, or by players outside the application's immediate scope of control. Players can be dynamically bound/unbound to roles [1.5] as demands on the application change, as the players' performance varies, or as the environment changes. A player may play more than one role, but at any point in time an instance of a role is always the responsibility of a single player (there can be many instances of the same role type in an organisation [1.4]). Roles may be temporarily unfilled by players [1.8]. In ROAD, these roles are first-class runtime entities that can be played by various players at different times. Roles, as performed by their players, satisfy their responsibilities to the organisation as a whole. *Functional roles* (as distinct from the *organiser* roles discussed below) are focused on first-order goals; that is, on achieving the desired application-domain output. Functional roles and their players constitute the process, as opposed to the control, of the system. The difference between a role and a player is that functional roles define an abstract function or a 'position' within an organisation, while role-players "do the work". This is in clear contrast to the usual concept of roles in object-oriented modelling, where a role is a descriptor for one end of a relationship. In Chapter 5 we examine in more detail, roles, players and their relationship.

Contracts associate two roles. They also monitor and regulate interactions between the roles. As all roles (as opposed to players) are internal to the organisation, ROAD contracts are also internal to the organisation, unlike inter-organisational service-level agreements. Like roles, instances of contracts are runtime entities. All runtime communication between functional players bound to the organisation is *via* contracted roles. If necessary, contracts intercept the communications between roles. Contracts define the mutual obligations of the participant roles in an organisational context. They define what interactions are permissible or required by the participant roles, and can be used to enforce sequences of interactions. Contracts can also be used to set arbitrary performance conditions on the roles' interactions [2.3-2.6], and monitor those interactions for compliance to those conditions [2.1]. Contracts thus encapsulate both the coordination and the performance management of interactions. Figure 4-1 below illustrates the relationships between roles, players and contracts. Contracts are further discussed in Chapter 6.

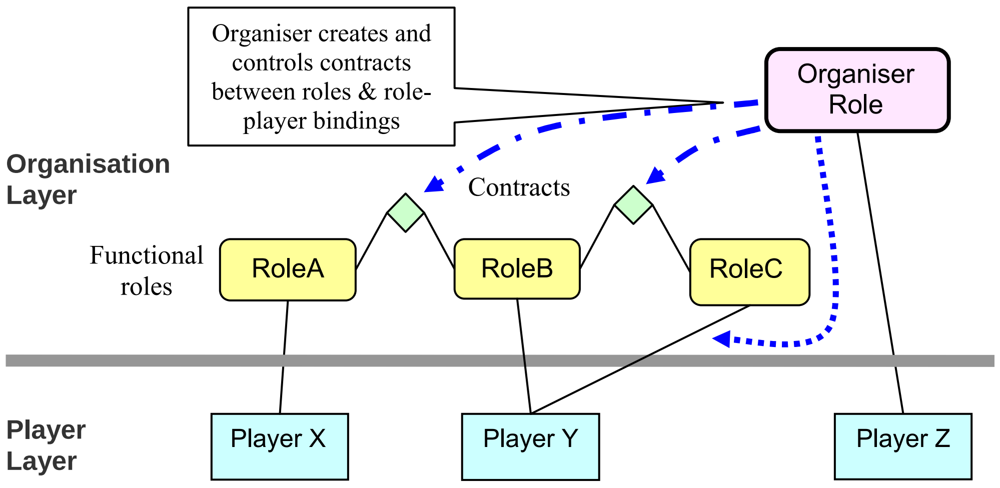


Figure 4-1: Roles, contracts and organisers from an organisation layer separate from players

Organisers [3.2] create and destroy roles. They make and break the bindings between organisational roles and players (player selection), and create and revoke the contracts between the roles. They can thereby create various configurations of roles and players [1.1]. Organisers set performance requirements for the contracts they control, and receive performance information from those contracts. Organisers have reconfiguration strategies they can employ if they detect under-performance in the composite they control [3.1]. Organisers are themselves a role-player pair, so that various role-players (e.g. Player Z in Figure 4-1) can be dynamically bound to the organiser role [3.9-3.11]. These organiser role-players may be of varying capability. In short, organisers provide the adaptivity to the application by managing the indirection of association and instantiation (as discussed in Chapter 2). Organisers, along with the contracts and roles that they control, can be viewed as a management layer that composes and controls the interaction of the functional role-players [3.6]. This management layer can be superimposed on to pre-existing players/components [3.3, 3.6, 3.8].

Each organiser is responsible for the configuration of a set of roles and contracts. We call such configurations *self-managed composites*. Each self-management composite has an interface (membrane) that defines its potential interactions with the external environment, and exactly one organiser role [3.4] that manages it internally. We call these regulated clusters of roles “self-managed composites” because each composite attempts to maintain a homeostatic relationship with its environment and other composites. In terms of a management analogy, a self-managed composite in a business organisation would be a department (e.g. manufacturing department).

Such managed composites are themselves role-players that perform definable domain functions (roles) within higher-level composites [1.3]. A role-based organisation is built from a recursive structure of self-managed composites. This structure is coordinated through a network that connects the organiser roles of each of the composites. The network of organiser roles and the contracts they control constitute a regulatory management-system. Organisers, self-managed composites and the management system are discussed in detail in Chapter 7.

4.2. Expository example

To illustrate how role-based coordination can be used to create adaptive software systems, we will use the example of a mixed-initiative (Horvitz, 1999) automated manufacturing production system. Let us consider a highly simplified manufacturing department that makes widgets. This department has an organisational structure consisting of a number of different roles that perform different functions (the rounded rectangles in Figure 4-2 below). These roles are Foreman, ThingyMaker, DooverMaker and Assembler (who assembles thingies and doovers into widgets). The Foreman’s role is to supervise ThingyMakers, DooverMakers and Assemblers and to allocate work to them. The WidgetDepartment also has a manager role (an organiser) that is responsible for creating roles in the department, for creating the associations between the various functional roles, and for assigning entities to play those roles. The WidgetDepartment is, therefore, a composite of these roles and the players who perform them.

Contracts associate roles. The only way two roles can communicate is via a contract. For example, the mutual obligations of roles of type Foreman and ThingyMaker are captured in a Foreman-ThingyMaker contract type. As shown in Figure 4-2, the control relationship between a Foreman and a ThingyMaker conforms to a Supervisor-Subordinate pattern, with the Foreman being the supervisor of the subordinate ThingyMaker. In ROAD this reuse of control patterns is implemented by having the Foreman-ThingyMaker contract class inherit its interaction patterns from an abstract Supervisor-Subordinate contract. As systems become more open, the components, agents or services that play roles are not necessarily well-tested or well-trusted. Interactions between roles (as player proxies) therefore need to be actively controlled. The runtime control that a ROAD contract imposes is a way of ensuring that interactions between the roles are appropriate to the organisational requirements, and that the players bound to the organisation are well-behaved.

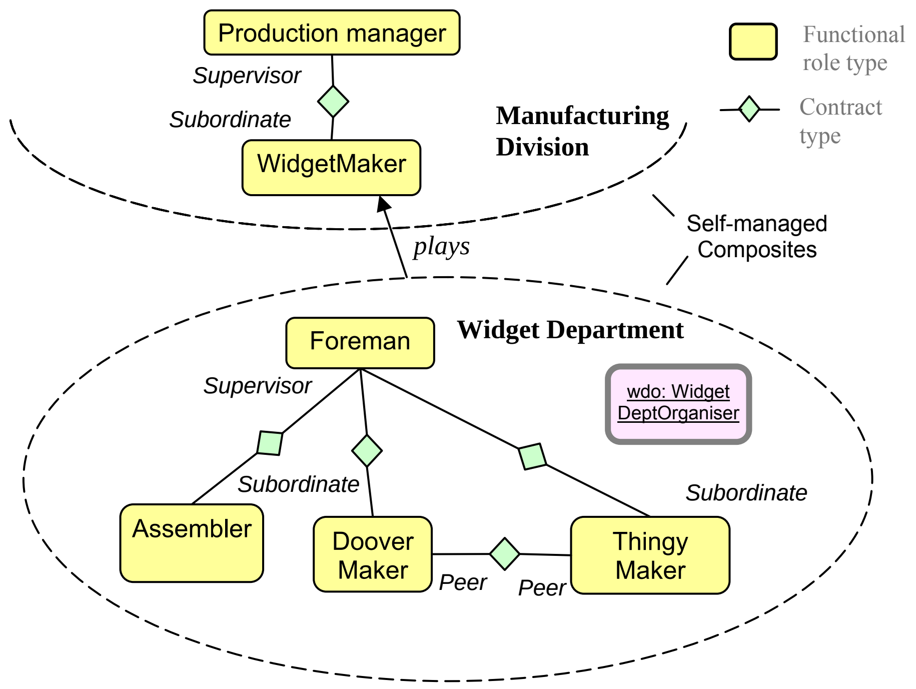


Figure 4-2: Organisational chart of Widget Department

The organisational chart in Figure 4-2 is of an *abstract* organisation, in that it shows classes and class relationships, rather than instances of roles, contracts or players. When a composite organisation is instantiated, instances of the roles and contracts are created, and player instances are bound to (some or all) of those role instances. A diagram of an instantiated WidgetDepatment (wd) is shown in Figure 4-3 below.

The roles in our WidgetDepartment can be performed by a variety of heterogeneous players. Players can be software objects, components, agents, external services, machine controllers, or humans interacting with the application through a user interface. Or players can themselves be composites of roles and players. In our example, the thingies are produced by machines that interact with the application through controller components; doovers are outsourced from an external supplier via a Web Service interface; assembler players are interactive UI components that require human employees, who do the physical assembly, to record their work; and the foreman is an automatic work scheduling component that has been provided by the Company's legacy scheduling software. This WidgetDepartment (wd) is itself a role-player that plays the WidgetMaker role instance (wm) within the broader ManufacturingDivision of the Company. The WidgetDepartment composite-player only interacts at a functional level with its WidgetMaker role in this context (blind communication [1.7]). Note that, as a player, the WidgetDepartment is a fully separable component from the enclosing Manufacturing Division composite, and might always be replaced by another player that can fulfil the WidgetMaker role.

In Figure 4-3, the functional and non-functional requirements of the WidgetMaker role are defined in contract C1. The WidgetDepartment's organiser-role (wdo), that is played by op1, creates, monitors and controls the contracts (C2a, C2b, C3, …) between functional roles (f, tm1, tm2, a1, …), and binds players (p1, p2, p3, …) to those roles.

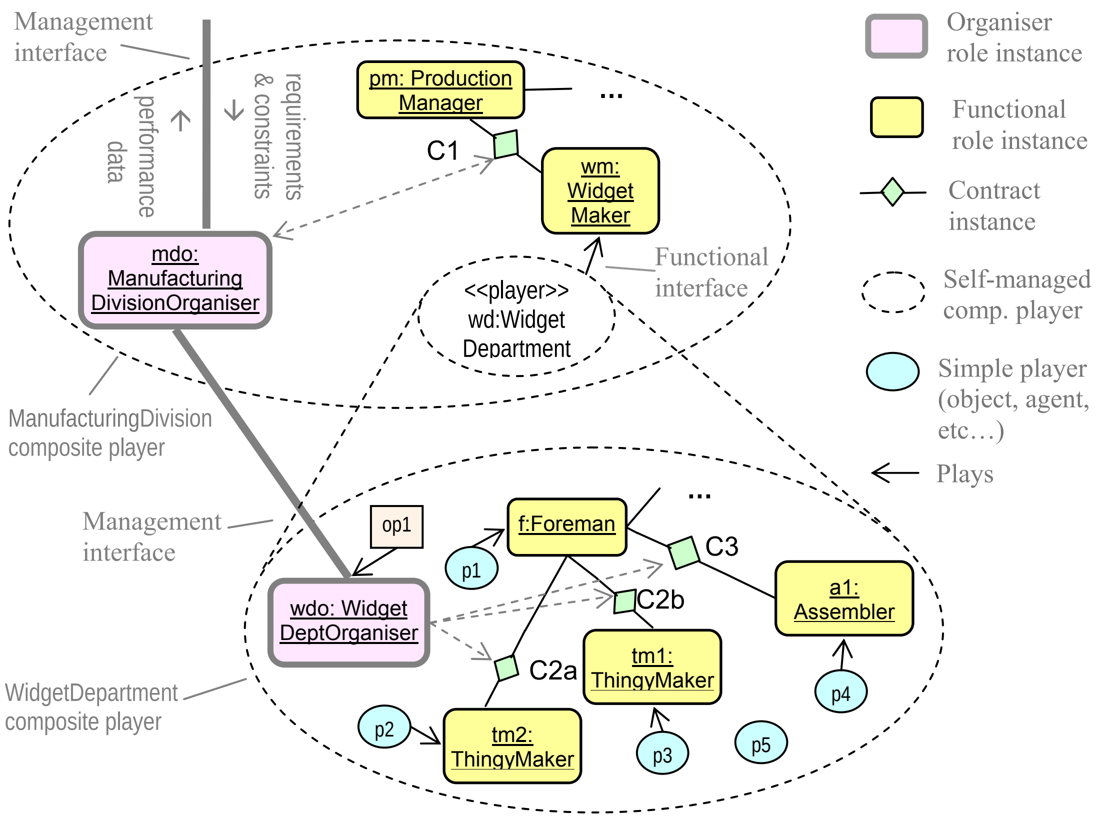


Figure 4-3: Exploded view of nested self-managed composites (not all roles and players shown)

Players will vary in their capability to perform a role. Capability has both functional and non-functional aspects. To be able to perform the role at all, the player must meet the functional requirements of the role; that is, it must be able to meet the goals and perform the tasks allocated to the role. But the player will also have to meet non-functional requirements such as speed, accuracy, reliability and so on. In conventional object-oriented design, all objects of the same type are treated as having identical capability and behaviour. However, in a more open system such as our WidgetDepartment, we cannot assume that all players of, say, the role ThingyMaker will have the same capabilities. For example, some types of ThingyMaker machine may be faster, more accurate, or more costly than others. Even if the machine-players are of the same type, instances of that type may be running in different contexts that affect their relative performance with respect to their roles. One player may be better connected, better resourced, better maintained, available more often, etc. than other players of the same type.

An adaptive system must respond to changes in the requirements of the system, or changes in the environment, or both. The Widget Department needs to be able to cope both with changes in the demand for its widgets, and with changes in the capabilities/availability of its players. For example, orders flowing into the department to make new widgets might increase, such that they exceed the capacity of the department to manufacture them. Or, one of the ThingyMaker players may become unavailable, unreliable or too costly. It follows that there must be a representation of the non-functional requirements of each of the roles in the Widget Department, and a way of measuring whether or not the players of those roles are meeting those requirements.

As a role-based organisation structure, the Widget Department has two basic adaptive strategies available. Firstly, it can restructure the relationships between the roles in the Department. For example, an additional ThingyMaker role (and player) could be added to cope with an increase in demand as shown in Figure 4-3 (i.e. non-functional restructuring [1.4]). The second strategy is to replace a player with an alternative player that better matches the required capability (e.g. player p5). For example, the production of thingy parts might be outsourced to a third-party service. The appropriateness of these adaptation strategies needs to be evaluated in order to mitigate any gap between required and actual performance. The organiser role must also have a means of enacting that strategy.

A ROAD application is built from a recursive structure of self-managed composites. Apart from the functional interactions that flow through contract and roles, this structure is coordinated through a network that connects the organiser roles of each of the composites. This network of organiser roles constitutes a management-system that is separate from the functional system. In our manufacturing business, information on non-functional requirements, capacity and constraints (e.g. financial constraints) flows over this management network. Depending on the capability of organiser players, this information can be used to plan changes to the system (VSM model's Adaptation and Planning subsystem (System 4) as discussed in Chapter 2).

* * * * *

In summary, there are a number of capabilities that a role-based system needs if it is to be adaptive at runtime to both changes in requirements and to changes in the operating environment. These capabilities include the ability to represent requirements; to measure the performance relative to those requirements; to evaluate strategies for adaptation; to restructure relationships between roles; to select appropriate players for those roles; and the ability to control the interactions between players via those roles. In the following chapters we will show how the ROAD framework meets these required capabilities.

5

Roles and Players

In its general usage the concept of a *role* defines the relationships of an individual within a particular social context (Steimann, 2000). In this thesis, we are concerned with software systems in which social contexts are intentionally designed and structured. As discussed in Chapter 2, we call such contexts *organisations*, where organisation refers to both the relationship of roles in the system, and the processes that maintain the viability of these relationships in response to changing goals and changing environments. Roles are the nodes of designed organisational structures.

In this chapter we define the characteristics of ROAD organisational roles and players, and contrast ROAD roles with other views of roles found in the literature on software modelling and design. The chapter provides a rationale for our approach to roles, and is structured as follows. Section 5.2 is a brief discussion of the various conceptions of roles in software development methodologies. In particular, we examine whether methodologies use roles merely as an analysis/design concept, or whether the role is reified as an implementation entity. We argue that to create adaptive role-based software organisations, roles need to be reified. Section 5.3 addresses the following questions. If roles are implementation entities, to what degree are roles and their players separate? Do roles have an independent identity? Can a role exist independently from the role player? We will briefly examine the various methodological responses to these questions, and point to the need to radically separate roles from players in organisational structures. We propose that role identity should be organisation-centric, rather than player-centric. In Section 5.4 we define the essential properties of both roles and players within an adaptive organisation. If we want to design and implement an explicit organisational structure, what properties will be needed in the roles that make up that structure? Consequently, what general properties will be needed by the players who play such organisational roles? We then introduce some issues that arise from the radical separation of roles and players. The subsequent sections of the chapter address these issues. In Section 5.5 we propose a novel conceptual framework based on the autonomy permitted by the role and the capability of the player. Players in different roles within a software organisation may be very heterogeneous (objects, components, services, agents or humans), and have very different degrees of autonomy and capability. We then examine various implementation strategies for defining role-players with various levels of autonomy. Possible solutions to this problem are discussed and related to object-oriented and agent-oriented approaches. Section 5.6 proposes that players should use "blind communication" in order to make organisations more adaptable through maintaining the separation of structure and process. Section 5.7 discusses the problem of the preservation of state in role-based organisations.

5.1. Roles as design and implementation entities

Roles are a recurring concept in both object-oriented and agent-oriented methodologies, not to mention data-modeling (Steimann, 2005). In these approaches roles may appear as concepts at the analysis and design stages, but are not necessarily implementation entities as they are in ROAD.

In conventional object-oriented methods, roles have figured as an annotation on the relationship between objects. In UML, roles are a descriptor for the ends of an association between classes (the concept of role has been subsumed by the concept of ConnectorEnd in UML 2.0 (Object Management Group, 2004)). In some methods, such as OOram (Reenskaug, 1996), roles are central concepts to the analysis and design. In OOram roles are nodes in an interaction structure (role-model). These role-models can be based on any suitable separation of concerns. Responsibility-driven design (RDD) also focuses on collaborations between roles, but such contracts between roles are seen as "really meaningful only in the programmer's mind" (Wirfs-Brock and McKean, 2002). In such approaches, roles are used in the modelling and to inform the design, but disappear as entities during implementation.

Other approaches based on role and associative modelling define roles as first-class design and implementation entities (Kendall, 1999a; Kristensen and Osterbye, 1996; Lee and Bae, 2002; Fowler, 1997; Bäumer, Riehle, Siberski et al., 2000).

Fowler (1997) discusses the implementation of roles in object-oriented design using a variety of object-oriented patterns. Kendall (1999b) has shown how program aspects can be used to introduce role-behaviour to objects. In Kendall's approach, roles are encapsulated in aspects that are woven into the class structure. Such approaches see roles as encapsulated implementation entities, but they vary as to whether roles can exist independently from the objects that play them. In common with the approach in this thesis, a number of object-oriented frameworks and languages that treat roles as first class entities have been developed (Baldoni, Boella and van der Torre, 2005b; Colman and Han, 2005a; Herrmann, 2002). These are discussed in more detail below.

Roles also figure in a number of agent-oriented approaches (Juan, Pearce and Sterling, 2002; Ferber and Gutknecht, 1998; Odell, Parunak, Brueckner et al., 2003; Zambonelli, Jennings and Wooldridge, 2000). Gaia in particular, (Zambonelli, Jennings and Wooldridge, 2003) extends the concept of a role model to an organisational model. Like some object-oriented approaches, roles are not an implementation entity. For example, in Gaia role models are developed at the analysis and architectural design stages, but roles are mapped to agents (not necessarily on a one-to-one basis) during the detailed design stage. Other agent-based models (Odell, Parunak, Brueckner et al., 2004) see roles as a key modelling concept but, being implementation-independent, these models give no indication of how these roles are to be realised. In general, if an agent-oriented methodology only has agents available as implementation entities, then it will lack the expressiveness to explicitly represent roles in an organisational structure.

As our aim is to create explicit organisational structures at the code level, we require roles that can be created and manipulated as implementation entities.

5.2. Two perspectives on roles – player and organisation

Kristensen (1996) defines the characteristics of roles in object-oriented modelling. These include:

- *Dependency*: a role cannot exist without an object
- *Identity*: an object and its role have the one identity
- *Visibility*: the visibility and accessibility of the object (a.k.a. player) is restricted by the role
- *Dynamicity*: a role may be added or removed during the lifetime of an object

- *Multiplicity*: several instances of a role may exist for an object at the same time
- *Abstractivity*: roles can be classified and organised into generalisation and aggregation hierarchies.

The above characterisation of a role has been widely adopted in the object-oriented literature, if not object-oriented practice. Roles can be implemented as runtime entities and yet have no independent identity or separate existence from their players. For example, Kendall (1999a) and Kristensen (1996) encapsulate roles as implementation entities but allow them no existence separate to the objects to which they are bound. While roles exist as a class, they can only be instantiated when bound to an object. Steimann (2000) provides a useful overview of approaches where roles are seen as adjuncts to object instances. Roles are seen as clusters of extrinsic members of an object. Such roles are carriers of role-specific state and behaviour but do not have an identity.

All the above approaches are object-centric or *player-centric*. The object is seen as the stable entity to which transient roles are attached. The identity of the role is an adjunct to the identity of the object. The role of an object is not an independent entity, but its appearance in a given context (Steimann, 2000).

An alternative perspective, which is adopted in this thesis as well as in Baldoni (2005b) and Herrmann (2002), is to look at roles from an *organisation-centric* viewpoint. From this perspective, a role's identity and existence derives from the organisation that defines the roles and associations – not from the player itself. This dependency of a role on a group is also apparent in some agent-oriented approaches (Odell, Nodine and Levy, 2005). Roles are the more stable entity within the organisational structure and transient players are attached to these roles. A role instance may be played by different players at different times (although not simultaneously). In an organisation, there is generally no restriction on a single player playing multiple roles. These two perspectives are illustrated in Figure 5-1 below.

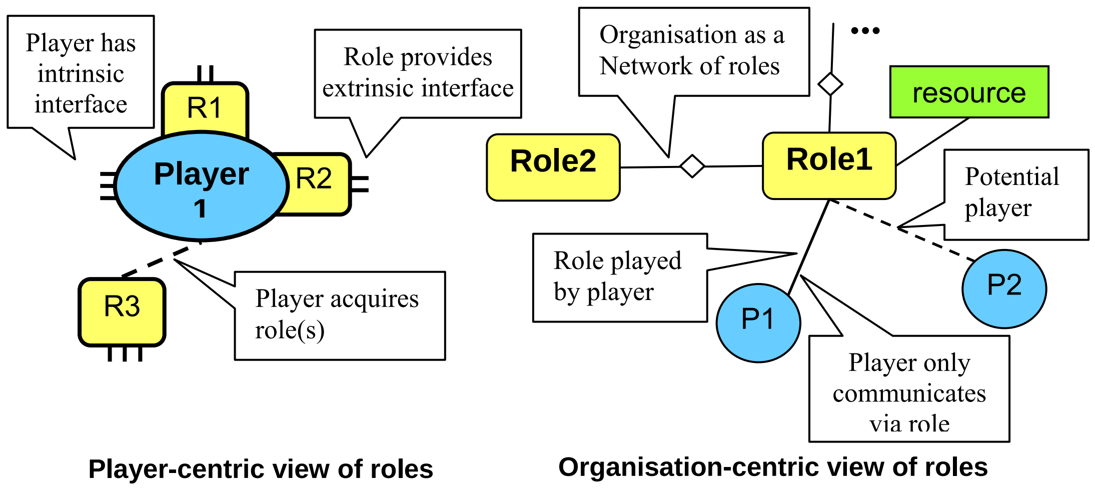

Player has intrinsic interface
R1
Player 1
R2
Role provides extrinsic interface
R3
Player acquires role(s)
Player-centric view of roles
Organisation as a Network of roles
•••
resource
Role2
Role1
Potential player
Role played by player
P1
P2
Player only communicates via role
Organisation-centric view of roles

Figure 5-1: Player-centric and Organisation-centric perspectives on roles

The *organisation-centric* view of roles accords more with the characteristics of roles in human organisations, such as a bureaucracy or a business. We call such roles *functional roles*, as they are part of the domain process (as distinct from management) of the organisation. The network of these roles is the basis of the organisational structure. If functional roles are nodes in an organisational structure, then a role may have associations a number of other roles of various role-types. A functional role may therefore consist of a number of interfaces – one for each of its associations with other roles. This is a different view from a conventional object-oriented view of a role: as a descriptor for one end of a single association[1].

A functional role instance is a "position" to be filled by a player (Odell, Nodine and Levy, 2005). There may be multiple role instances of the same type within an organisation. Role instances may be temporarily unassigned to players. For example, if an employee (player) resigns from their role as Production Manager within a manufacturing business, the role does not cease to exist. That position (role) within the company structure may be temporarily unassigned, but the company may continue to function viably in the short term. Orders can still be taken, current work orders still be manufactured, finished goods can still be shipped, accounts can still be processed and so on. Nor does the identity of the role depend on the identity of the player. From the organisation's point of view it does not matter whether employee John Doe or Jane Smith performs the role of Production Manager as long as they both have sufficient capability. In a viable organisation the role model (organisational structure) is not just a design concept that helps structure the relationships between employees (players). It is also a set of relationships between roles that is maintained and manipulated (to some degree) independently from the players that are assigned to those roles. The ability to dynamically bind different players to a role gives the organisation a degree of adaptability in meeting changing goals and environments.

Organisational roles therefore can be thought of as having a number of states in relation to the binding with a player. An instance of a role can either be assigned a player or left unassigned. As Odell et al. (2003) point out, the relationship between a role and an assigned agent may also be in an active or suspended state (e.g. our Production Manager has gone to lunch and although she is not active in her role, she still occupies that position).

To summarise the characteristics of functional roles within an organisation, Kristensen's characteristics of *Dependency* and *Identity* do not hold (the other characteristics are still applicable). We can modify Kristensen's characteristics of roles as follows:

- *Existence independent of player*: Role instances in an organisation do not depend on players for their existence. They depend on the organisation for their existence.
- *Independent identity*: Role instances have an organisational identity that is independent from their players even though the role and player act as a unity within the organisation.

The separation of organisational roles from the entities that play them, allows the definition of abstract organisational structures that are independent of particular players. Such a structure in a human organisation would be described, for example, in a company's organisational chart where the nodes are the roles in the company and the arcs are the authority relationships. However, the radical separation of roles from role-players introduces the problems of how to define the dividing line between extrinsic (role) and intrinsic (player) properties in the combined role playing entity, and how to preserve the integrity of the organisation's processes when players are swapped. We address these issues in the following sections.

[1] ROAD *does* characterise the ends of associations between functional roles as "performative" roles. However, because these roles are properties of the functional role-role association, we discuss them in the next chapter on contracts.

5.3. The properties of roles and players in adaptable organisations

In ROAD, we conceive of the role as expressing a function that serves some purpose in the organisation. It defines what the role-player needs to do at some level of abstraction, and defines the provided and required relationships with other roles. It may also define a relationship to tools and resources. The player, on the other hand, initiates actions in line with its capability to perform the defined role. The player has intrinsic properties that gives it this capability. This concept of the intrinsic nature of player capability is common to both object-oriented (e.g. the "core object" in (Kristensen and Osterbye, 1996)), and agent-oriented (e.g. "agent physical classifier" in (Odell, Nodine and Levy, 2005)) approaches.

Let us illustrate the separation of properties between a role and a role player with an example from an organisation made up of people – a coffee shop business that has a role of coffee-maker[2]. The organisational context (the coffee-shop business) of this role is illustrated in Figure 5-2 below

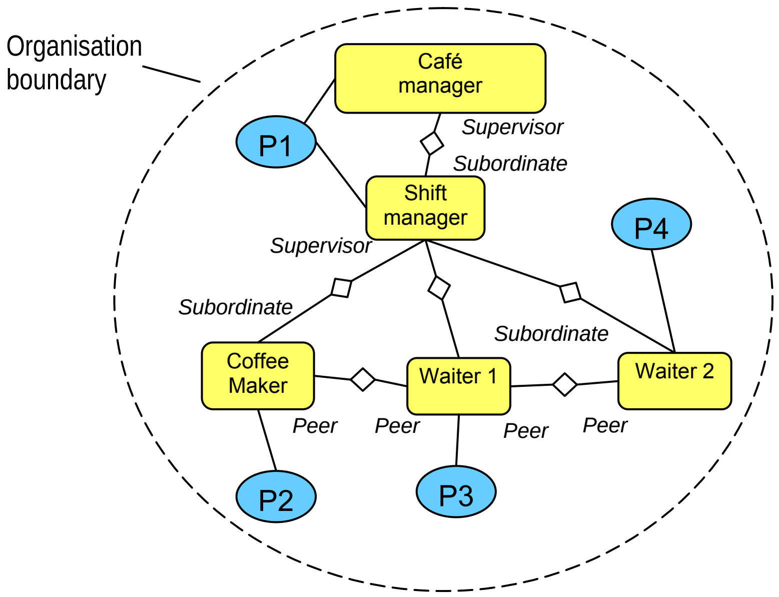


Figure 5-2: Organisational Chart with players

The coffee-maker role might be defined as follows. The goal of the coffee-maker role is to make quality coffee to a certain standard within certain time constraints, in response to requests from waiting staff. The role defines work instructions for preparing the coffee to the business' standard. The role also gives access to resources such as the expresso machine and ingredients, and it defines

[2] We temporarily depart from our Widget Department example, because the author knows more about making coffee than he knows about making thingies or widgets.

functional relationships with other roles in terms of what is provided and required. It also defines authority relationships with other roles in the business. For example, the coffee-maker is subordinate to the shift manager, peer to the waiting staff and so on. These authority relationships define the valid types of control communication that can pass between actors playing the respective roles (e.g. unlike a shift manager, a waiter cannot tell a coffee maker to finish their shift). A role has (explicitly or implicitly) a "position description" that describes the capabilities needed of a player assigned that role. In order to effectively play the role of coffee-maker, an employee (or rather a person playing the role of employee which is itself a generalisation of the coffee-maker role) needs to be able to follow work instructions, use the tools provided to transform the ingredients as required, and communicate with other role players following the conventions imposed by the authority relationships between the role types.

In general, an organisational role has the following properties.

- The *function* of the role expressed in terms of purpose, system-state, or process descriptions (depending on how detailed the level of prescription in the role and how much autonomy the player is able to exercise).
- *Performance requirements* for executing its functions are a property of the relationship of the role with its enclosing organisation, rather than an essential property of the role itself. Such performance criteria are set by the organisation. Actual performance of a role is always an externally measured property of an assigned role (player-role pair) because different players may have varying capability in performing the role. As performance of a role-player pair is always *in relation* to its organisation, it is therefore appropriate to represent performance as the property of a relationship with the organisation, perhaps as represented by other roles in that organisation. In our ROAD framework, the performance level with respect to non-functional requirements is recorded in contracts that associate roles (as described in the next chapter).
- *Interaction protocols* and *authority relationships* (power, expectations and obligations) with respect to other roles within the organisation and with external entities with which the organisation has associations.
- Access to, and restrictions on, *resources* controlled or owned by the organisation.

The above properties are fundamentally properties of the role's relationship with its organisation, i.e. its composite, other roles, and resources in the organisation. The role description is an aggregation of the properties derived from its relationships. From the perspective of a role player, this aggregate description is a definition of the knowledge and skills required in a player to enable performance of the role function. This is an interface definition with both functional and non-functional requirements. A role player needs to be able to execute the function defined in this interface at the specified level of performance and while meeting any other non-functional requirements it defines.

A role and its player *act* as a unity within the organisation, even though roles within an organisation have an existence and identity independent from their players. If we separate functional roles from the players who play them, what properties are ascribed to the role and what are the properties of the player? In particular, if we are to implement roles as first-class entities in a runtime organisation, a number of issues arise from the radical separation of roles and players. These issues relate to the division of responsibility between roles and players, namely:

- What level of autonomy do players have in fulfilling their role? Do roles "do" anything?
- Does knowledge of the organisational structure reside in the role, player or both?
- Are the roles or are the players responsible for maintaining state within the organisation?

The following sections discuss these issues and various strategies for the implementation of roles. The purpose of this discussion is to evaluate some of the alternative ways organisational roles might be implemented, in order to provide a rationale for the way roles have been implemented in ROAD. In particular, we will (where appropriate) illustrate these issues by comparing and contrasting how they have been implemented in three different approaches to creating role-oriented software organisations. These are powerJava (Baldoni, Boella and van der Torre, 2005c; Baldoni, Boella and van der Torre, 2005b), ObjectTeams (Herrmann, 2002; Herrmann, 2005), and our Role-Oriented Adaptive Design (ROAD) framework.

5.4. Levels of player autonomy

Complex systems can comprise heterogeneous players with varying capabilities. Such players operationalise the requirements defined in the roles. The level of operational detail at which these requirements are expressed may vary depending on the amount of autonomy that the organisation allows the player, and consequently on the capability of the player to act with some appropriate level of autonomy. Returning to our example of a coffee-maker, the work instructions for making coffee may vary in terms of their level of detail. Inexperienced coffee-makers may require detailed instructions on how to make a cup of coffee, while an experienced and capable coffee-maker may not need to follow instructions defined by the role but may just be given a system-state ("strong cappuccino"). This experienced coffee-maker may alter the process depending on the inputs ("the coffee-maker perceives the beans to be a darker roast than usual").

As can be seen from the above example, the granularity of the descriptions of the task contained in a role may vary depending on the autonomy granted to the player. Intentional action can be described at various levels of abstraction on a means-end (intentional) hierarchy as shown in Figure 5-3 below.

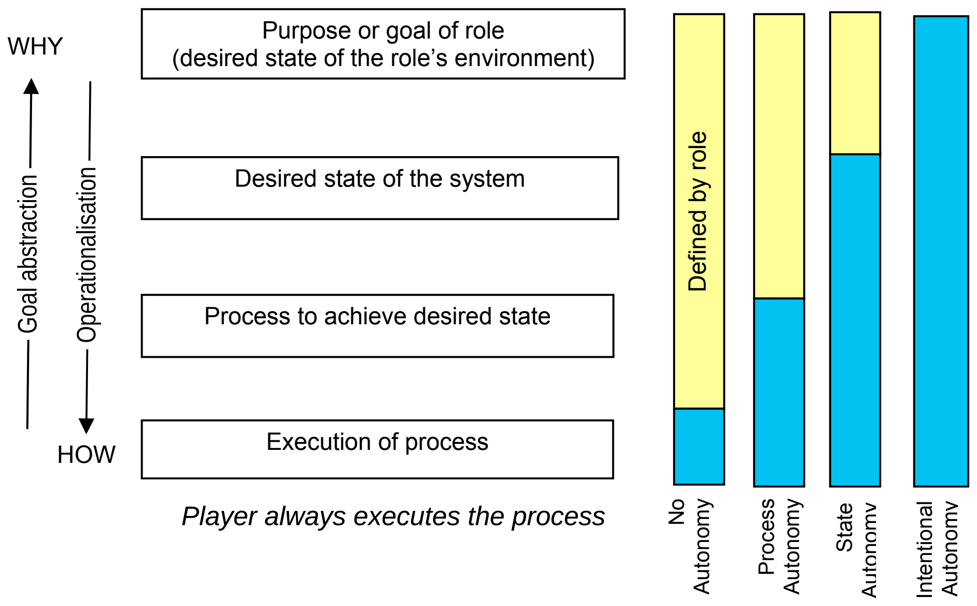


Figure 5-3: Shifting boundary between roles and players on an intentional hierarchy

In an intentional hierarchy, goals are operationalised at successively lower levels of abstraction, while the purpose of a function at one level of abstraction can be determined by referring to higher levels of abstraction. At its most abstract level, the role to be performed may be described by a *purpose* or *goal* in the environment external to the role or organisation ("keep the customer's happy by making good coffee"). At a more detailed level, the means to achieving the goal of the system might be described as a *state* of the system itself ("make a coffee to standard X"). At the next level of operationalisation, the *process* for achieving that state would be described by the role ("follow the coffee-making work-instructions"). Such work instructions could then be described by the role at progressively more detailed granularity.

At some point the atomic goals / states / processes must be interpreted and executed by the player. In this sense, the relationship between role and player is more like the relationship between program and abstract machine, than between two components at the same level of abstraction. In an organisational context, the amount of autonomy that a player can exercise in a role is defined by the organisation rather than the player. Based on the above intentional hierarchy, we can identify five levels of player autonomy. Ordered in terms of increasing player autonomy, these are:

1. **No autonomy –** player executes the process defined in the role
2. **Process autonomy –** player can choose the process to meet the system state defined in the role
3. **System-state autonomy –** player can choose a system state and processes to fulfil a goal defined by the role
4. **Intentional autonomy –** player can choose if it will fulfil a goal/state/process defined by the role

An additional level of autonomy can be identified, although it is not one defined by the role, that is:

5. **Autonomy from constraints –** player can violate constraints defined in the role.

As the role is organisationally defined, the intention or purpose of the role is always defined external to the player (i.e. it is implicit in the role itself). On the other hand, the process is always executed by the player. In this conception, unlike the player-centric view of role, the role does *not* execute any domain function. In the following sections we will discuss each of these levels of autonomy and possible strategies for implementing such roles in software.

5.4.1. Players with no autonomy

A player with no autonomy is told what actions to execute and always attempts to execute them. In our coffee shop example, the role contains detailed work instructions that define the process to be followed when executing a role task. The purpose and the system-state of the role are implicit as indicated by the dotted boxes in Figure 5-4 below. The implicit purpose and system-state have been reified into the process instructions by the role designer. In a static role definition there is no runtime translation from the role's intention to system-state, and from the system-state to process.

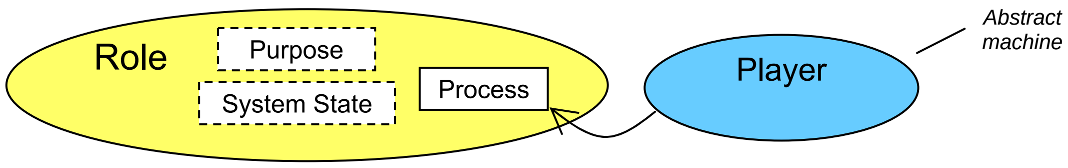


Figure 5-4: Alternative 1: Separation between role and player where player has no autonomy

However, the performance of the role can vary depending on the execution context provided by the player. For example, in our coffee shop, different employees are able to make coffee at different rates. Where the process is defined entirely in the role, as in Figure 5-4 above, the player can be viewed as an abstract machine that executes the process provided by the role. The role-player pair acts as a single entity executing in a particular environment. Such environments may have various computational characteristics. Roles performed by different players (computational contexts) may consequently have different observed performance. In terms of organisational dynamics, changing the role player is changing the machine on which the role is executed. The role identity does not change, nor does its functional relationship to the rest of the system change.

An alternative approach to implementing players that make no process decisions is to have the process interface defined in the role, but the process statically implemented in the player, as illustrated in Figure 5-5 below. This is the approach we have adopted in the ROAD framework. All domain-function is executed in the players. The role is an object that defines *required* and *provided* interfaces that express the properties defined above in Section 4. The role receives/sends messages from/to other associated roles via contracts; buffers in-coming messages (if a player is not currently active in the role); and delegates in-coming messages to the player. In ROAD, roles are always composed into self-managed composites under the control of an organiser. Power to act within the composite is conferred by its organiser who creates contracts between the roles.

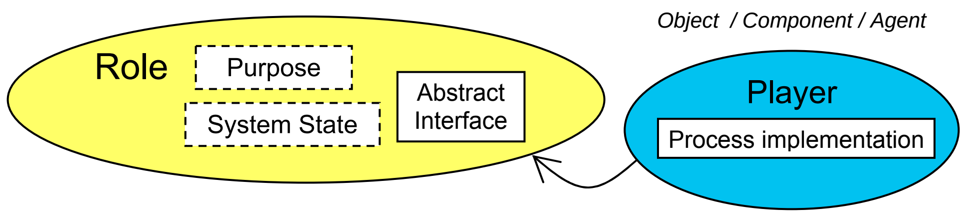


Figure 5-5: Alternative 2: Player as execution context and single process

Other role-based organisational approaches allow the splitting of domain-function between the role and the player. powerJava (Baldoni, Boella and van der Torre, 2005a) extends the object-oriented paradigm and Java programming language with a pre-compiler to implement organisational roles. Institutions (like ROAD composites) define roles that are played by players. However, in powerJava, unlike ROAD, roles themselves perform domain-functions and institutions maintain domain state. Institutions give 'powers' to the object playing the roles, rather than having roles statically defined within an institution. Likewise, in Object Teams (Herrmann, 2002) domain-function can be split between a role and a player (base-object). However, Object Teams does not support adaptivity through indirection of instantiation: once a role-object is created the link to its base-object (player) cannot be changed.

An advantage of having all domain-processes defined in the player is that the role structure (organisational composite) remains a purely management abstraction. The player can be of any type (object, component, Web service, agent, or back-end of a user interface) as long as the player conforms to the role interface.

5.4.2. Players with process autonomy

A player with process autonomy is given a task to perform in the form of a *system-state*, but it has some autonomy in deciding what steps are executed in order to achieve that task. The player must have the ability to translate a *system-state* provided by the role into a *process* which it can execute. If this system-state is variable then the player may require deliberative capability to effectively perform this translation. On the other hand, translation from the role's *purpose* to *system-state* is implicit – that is, carried out by the programmer when the role is designed.

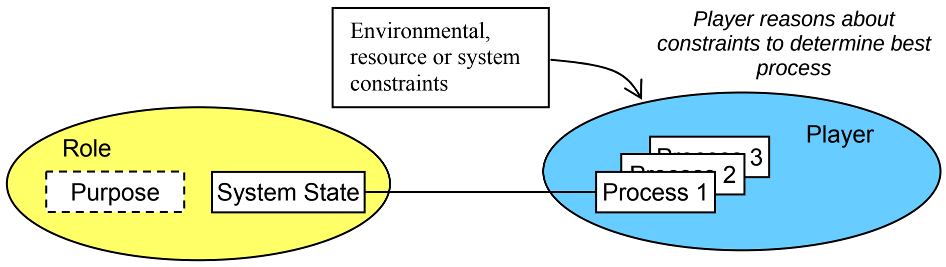

Environmental, resource or system constraints
Player reasons about constraints to determine best process
Role
Purpose
System State
Player
Process 3
Process 2
Process 1

Figure 5-6: A player with process autonomy must be able to translate a system-state to a process, then create or choose appropriate the process given constraints

Where a role provides only a system-state to be achieved, rather than a detailed process to be executed, the player must contain the process definition(s) to achieve the goal. From a viewpoint external to the player (that is, from the role's or organisation's perspective) the process is hidden, thus the player has apparent autonomy. If the role is subject to environmental perturbation (for example, changing availability of resources) the player may require some deliberative ability to decide what process is the most appropriate one to achieve that system-state.

Alternatively, rather than the process definitions being stored in the player, a role may also contain pre-defined process plans from which the player selects. Such a player might, for example, be implemented using a BDI agent with a range of plans that can be applied to differing situations and system-states. As with all other types of player, players with process autonomy are performing the role within a computational context that determines the performance of the role-player. Player performance cannot be fully characterised independent of their context, because they are situated entities. However, while actual performance is always related to a situated role-player pair, representations of both the role performance requirements, and the player performance capability, are probably necessary to enable the selection of appropriate players for particular roles.

5.4.3. System-state autonomy

A player with system-state autonomy is given an external goal which may be satisfied by a number of states. A software example of system-state autonomy would be an operating system that maintains processing capacity by deciding the run-time priority of processes. A number of states could satisfy this goal and the player must choose between them.

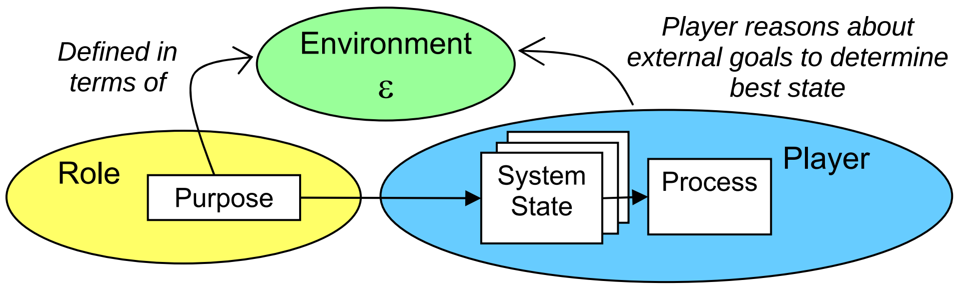


Figure 5-7: A player with system-state autonomy must be able to translate a role's purpose into a suitable system-state, and then into a process

At the top-most levels of an organisation, players may need to determine the appropriate system-states that best satisfy the role's purpose, given a range of variable internal and environmental constraints. In closed systems, where there is a manageable finite number of system-states (possibly pre-defined states or states defined by a limited set of parameters), it might be possible for a player to have the capability to evaluate these alternative states and select the one that best matches the role's goal. However, in more open environments, where there are a large number of constraints, it is difficult to automate such capability. Such a role would be typically played by a software developer at design-time, or by a human operator at run-time. These players need the perceptual capability to identify relevant constraints; to devise appropriate system-states; to be able to model the effect of various states on the role's purpose; to determine the best system-state by trading off costs and benefits; and to devise the processes to realise these states.

Such capability is often required of a human player interacting with a system in a supervisory role; for example, in a mixed-initiative control system (Horvitz, 1999). Providing a role interface to all players allows us to construct systems that have a consistent architecture based on roles, regardless of whether the players are fully automated machines, or are humans using a user interface. In many control systems, automated control can cope with anticipated perturbations. However, when unanticipated conditions occur, human operators must replace machines as the role players (Rasmussen, Pejtersen, and Goodstein, 1994). By abstracting roles from players, systems can be developed that better enable this transition.

5.4.4. Players with intentional autonomy

The purpose of the role is organisationally defined — it is implicit to the definition of the role. From the organisation's viewpoint its players should not exercise intentional[3] autonomy; i.e. they should not have the discretion to decide which organisational goal to adopt. A player with intentional autonomy is a free agent with the ability to decide whether or not to satisfy external goals — it has (or we ascribe to it) its own intentions. However, players may play roles in a number of social networks or organisations (or even multiple roles in the one organisation) which can lead to conflicts in priorities and the allocation of resources as shown in Figure 5-8 below.

A *cooperative* entity will fulfil the request if it can. A *competitive* entity only does so if it receives sufficient reward. In the software domain, proactive software agents might exhibit intentional autonomy. Cooperative agents attempt to collaborate to achieve system level goals, whereas competitive agents in a market-based system attempt to maximise their own utility.

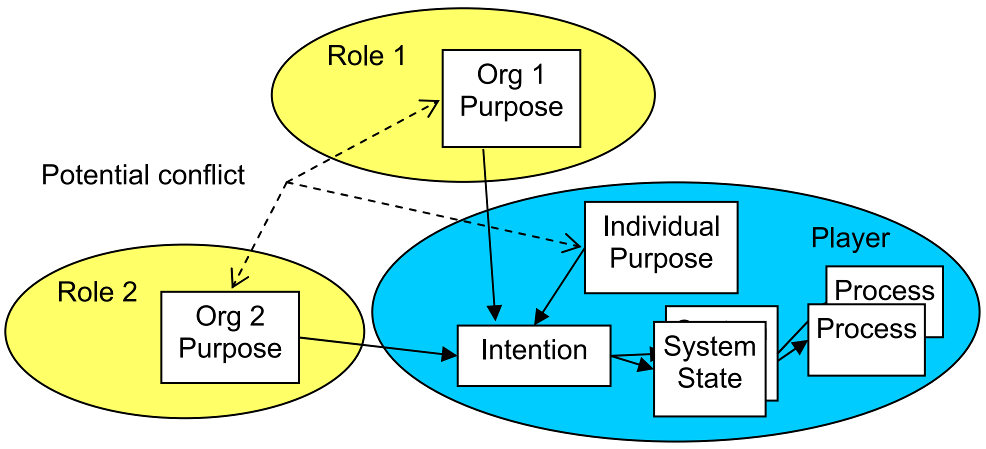


Figure 5-8: Organisation and individual purpose may have to be resolved by a player with intentional autonomy (a free agent)

In these cases, the players need some mechanism for prioritising these conflicting goals, and some way to form an intention to achieve a system-state. Negotiation between the player and the organisation about the level of service provision might also be necessary.

5.4.5. Players with constraint autonomy

A player that exhibits constraint autonomy is prepared to violate constraints, norms or even rules, in order to achieve its goals. For example, a greedy software agent may take no account of the computational resources it consumes. A malicious or anti-social agent may deliberately try to harm other agents or the system

[3] We use the word "intention" in the general sense to indicate goal-directed agency as in Dennett (1987) and Searle (1983), rather than in the limited BDI sense (Georgeff, Pell, Pollack et al., 2002) of

environment itself. Well run organisations should generally avoid assigning roles to greedy or malicious players. If the use of such players is unavoidable, their behaviour has to be tightly controlled. Badly behaved players that can exhibit autonomy from constraints might also be used in organisations provided all their interactions with the organisation are controlled. This is the case in the ROAD framework, where all player interaction is via their roles, and all interaction between roles is controlled by contracts. These contracts can be used to ensure the player does not violate organisational constraints. In addition, the contracts can monitor (although not enforce) the performance of the player.

5.4.6. Capabilities required of players with different levels of autonomy

Given this conceptual framework we can now define the generalised capabilities that are needed by players to exercise the level of autonomy defined by the organisational role.

Table 5-1: Level of capability needed for players with different level of autonomy

Level of Autonomy	**General Player Capability Needed**
No autonomy	Ability to communicate, follow instructions and effectively use tools and resources provided by the role. The instructions will be 'interpreted' by the player and then executed.
Process autonomy	above + ability to select appropriate processes and tools to complete prescribed tasks
System-state autonomy	above + ability to sense the environment, to determine which state best fulfils the goal defined by the role in the current environment given the tools and resource available
Intentional autonomy	Players of roles defined by a *closed* organisation do not have intentional autonomy with respect to their role. The intention of the role is defined by the organisation. However, conflict may arise if the player is playing more than one role. In a more *open* organisation (where players may belong to other organisations), conflicts may arise between competing goals. Such a 'free agent' player would need the ability to negotiate with the various organisations to which it belongs to try and achieve optimal outcomes.
Constraint autonomy	Players that exhibit constraint autonomy should not be bound to roles in organisations, unless appropriate constraints can be imposed on the interactions of the player with the organisation.

an "intention to act" i.e. the selection of a particular course of action. Intention in our usage is more like the "Desire" in BDI.

In (Colman and Han, 2005b) we argue that to create a viable organisational structure that can achieve system level goals in a complex environment, different role players *must* have varying degrees of autonomy. Players need capability commensurate with the complexity of their respective environments as defined by their role. Mintzberg (1983) has shown that in human organisations, the higher the role's level in the organisational structure, the less formalised and standardised the behaviour required of that role. The higher the role is in the hierarchy, the more autonomy and capability that player needs to be able to adapt to environmental perturbations. For example, players with no autonomy cannot be expected to cope with highly variable environments given only fixed work instructions. However, autonomy comes at a cost. While a player with system-state autonomy can always perform a highly routinised role, it is not an effective use of resources, particularly if the player has to perform computationally expensive scans of the environment.

Given the diversity of player types that can participate in a ROAD application, the ROAD framework does not define a general format for the interface between a role and a player. As a minimum *provided* and *required* functional interfaces need to be defined in a role's 'position description'. Any non-functional attributes will also need to be expressed in this interface. Much recent research has been done to define such "rich" interfaces, e.g. (Han, 1998; Han and Jin, 2005). Furthermore, if players (e.g. agents) are to be developed as general purpose role-executors, rather than being designed for particular roles, they will need the ability to read and interpret these role 'position descriptions'.

If heterogeneous players are used, the framework should be extensible so that a new type of player can be added. Adaptors will need to be appended to roles to convert the internal interface to the appropriate interface of the players as shown in Figure 5-9 below. For example, if the ROAD role is implemented as a standard Java object, and the player is a Web Service, then an adaptor that presents a WSDL (W3C, 2005) interface and converts between method calls and SOAP messages will be needed. In an agent context, another adaptor may need to be defined that can speak an agent communication language such as FIPA/ACL (FIPA, 2002).

5.5. The separation of organisational structure from process

Separating roles from the players that play those roles also allows us to define organisational role-structures that are separate from the players that perform the function of those roles. In conventional program structures, objects / components /

agents talk directly to each other. These mutual references can be hard-coded or they can be variable references that are dynamically set. However, such structures are fragile to the extent that all nodes must be present in the structure for it to be well formed. The representation of the structure is also implicit in the references that are embedded and hidden (if object-oriented principles are followed) in the components themselves. Such an approach also requires that the entity that is performing a function must have some representation of the structure in which it will participate; that is, a representation of the context in which the function will be used. This tangles structure and function in the code and inhibits adaptivity. If a component participates in multiple relationships, to change the component requires the restructuring of all those relationships.

An alternative approach has been called the principle of "blind communication" (Oreizy, Gorlick, Taylor et al., 1999). Lieberherr (1996) similarly proposes that "structure shy" components are a prerequisite for adaptive systems. In these approaches structure is defined separately from the components of that structure, and can be superimposed *a posterior* on those components. This is the approach adopted in ROAD, as the structure is defined by associating a role with one or more other roles by means of contracts. On the other hand, a role instance is always bound to one player at most. Having a single interface between a role and its player thus simplifies the substitution of players.

A consequence of having players that are structure-shy is that it is the roles (and the connectors/contracts that bind them) that hold a representation of their local structural relationships. A role can be associated with multiple roles but with only one player. While all incoming messages (from other roles) are passed to the player, out-going messages need to be passed to an appropriate associated role. Roles therefore can be regarded as message routers. In Figure 5-9 below, Role A passes all incoming messages to its player, but must allocate out-going messages to either Role B or C, depending on the message type (if B and C are of different types). Other allocation schemes are needed where structures include multiple roles that fulfil the same basic function (e.g. there is more than one coffee-maker in the organisation). For example, if Roles B and C are of the same type (they can both handle the outgoing message), then Role A might route the message to the role-player pair that has the better response time, better reliability or whatever other quality of performance is of interest.

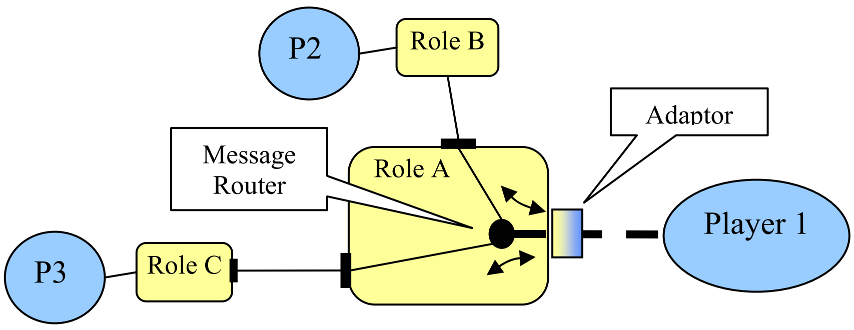


Figure 5-9: Role as message router - Principle of Blind Communication of Players

5.6. The preservation of state

Another consequence of the separation of a role and its player is the need to resolve the question of who is responsible for maintaining state. The maintenance of state is an issue because, in dynamic organisations, the integrity of the whole needs to be preserved even though the parts that hold state – the roles and the players – may change. We can distinguish two types of state that need to be maintained: communication state and domain state.

5.6.1. Communication state

In a ROAD organisation, communication between players is always mediated by their respective roles. Messages to a role may still be generated even though the role is temporarily unassigned or inactive, as described in Section 5.2 above. In order to be viable in the absence of players, organisations need to provide some form of message queuing and storage. The recipient player cannot be responsible for managing and storing these messages because that player may not always exist. A number of alternative approaches are possible to ensure the on-going viability of the organisation during the absence or transition of players. These include storing the message in the sending role, storing it in the receiving role (as shown in Figure 5-10), or, alternatively, having the organisation store outstanding messages in the contractual associations that connect roles. A further possible alternative of having the sending players hold the message request if the receiver is off-line is not be a good strategy as the sending player itself may become inactive. It also violates the principle of blind communication as the transmitting player would need to be aware of other roles and whether or not they have a player assigned.

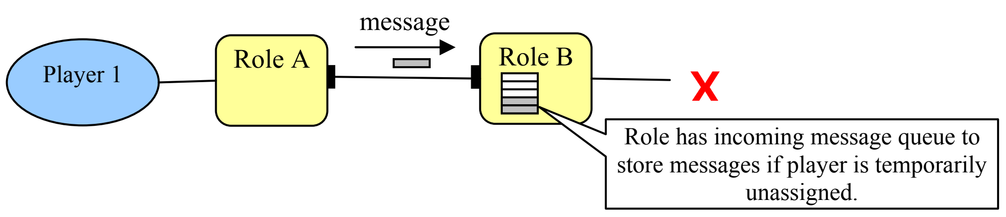

message
Player 1
Role A
Role B
X
Role has incoming message queue to store messages if player is temporarily unassigned.

Figure 5-10: Roles as messages buffers

While the ROAD framework includes message buffering in roles, the approach to how communication state is preserved is perhaps an implementation issue rather than a fundamental attribute of runtime role structures. The scheme employed for message buffering in the role is dependent on the mode of interaction (e.g. push or pull) and type of synchronisation used for transactions (e.g. asynchronous transactions). The integrity of message delivery might also be handled in a middleware layer. We address these implementation issues in Part III of the thesis.

5.6.2. Domain state

The other type of state that needs to be preserved, in the event of changing roles and players, is that of the state of the process being executed, i.e. the domain state. In object-oriented approaches, state is typically encapsulated in the objects. In other approaches, state is stored so that it is globally accessible. In the context of a role-oriented organisation, a number of alternatives exist as to where the domain state can be maintained. These are

- State is maintained in player
- State is maintained in role
- State is maintained in organisation

The *advantage* of having the domain state stored in the player is that it maintains the encapsulation of data and operations on that data (as in object-orientation). As it is always the player that operates on the data, the internal representation can be hidden and decoupled from the system as a whole. This results in the loose coupling of role and player, and facilitates the swapping of player implementation. The player only has to conform to the interface defined by the role.

The *disadvantage* of maintaining state in the player is that when a player is replaced any state relevant to the organisation must be transferred to the new player. Safe points also need to be defined (eg. between transactions), when it is permissible to swap players. Alternatively, roll-back or compensation mechanisms would need to be implemented. The problem of maintaining state during player transfer may be addressed by storing state either in the role or in the organisation composite itself. Storing state in the role does not entirely overcome the problem because, in an adaptive organisation, roles themselves are created and destroyed. Another approach, used for example in the powerJava framework (Baldoni, Boella and van der Torre, 2005a), stores state globally in the institution (the organisational composite). Access to such state (and resources) by roles is then controlled by the institution "empowering" roles. Such an approach may be beneficial for storing data related to the composite level of abstraction. However, if it is used to store state that is properly the responsibility of the player, such an approach would break the encapsulation of the player, leading to the well-known problems associated with global data (Parnas, 1972).

As the ROAD framework aims to create adaptive organisations, all domain state is maintained by the players in order to facilitate the swapping of players. As a consequence, it is necessary to either define stateless points in the execution process where it is safe to transition, or alternatively define methods for transferring state between players. How to maintain the integrity of the system during component interchange is an active area of research (e.g. see (Hillman and Warren, 2004) for an overview), but is outside the scope of this thesis.

5.7. Summary

Organisation is defined here as the relationships between *roles* in the system, and the processes that maintain the viability of these relationships in response to changing goals and changing environments. An organisation-centric view of roles sees roles as nodes in an organisational structure, rather than just behaviours that can be added to an object or agent. In an organisation, roles have an independent identity and existence from the players who are assigned to them. Roles are first-class runtime entities that can have a number of states with respect to players: assigned or unassigned; active or inactive.

The radical separation of roles and player raises a number of issues that need to be addressed. Organisational structures in complex systems require role-players with various levels of autonomy and capability. The relationship between a role and its player will vary depending on the level of autonomous action required of the player. A framework that supports such architectures needs to be able to handle bindings between roles and a diverse range of players (objects, agents, services, user interfaces etc.). The adaptable framework also needs to be extensible to handle new types of binding. In the ROAD framework players are “structure shy”. Consequently, roles need to act as message routers. The framework will also have to handle the problems of preservation of communication and domain state when the organisation is restructured. In a ROAD application, roles are stateful interfaces that preserve communication state if there are no players attached. Role players of various capabilities (including humans in some circumstances) can be dynamically assigned to roles as the demands on the system change, or the environment in which it operates changes.

The next chapter describes how ROAD roles are associated using contracts to create organisational structures.

6

Contracts between Roles

Our lives are governed by contracts. These contracts can be formal or informal. They include employment contracts, contracts of sale, business contracts, marriage contracts and so on. Even laws can be seen as a form of social contract between the citizen and the state. Contracts set out the mutual obligations of one party to another, and are concerned with the governance of interactions between the parties. Contracts are also a recurring theme in software development. As software becomes more distributed and open, the relationships between entities in the software system can become non-deterministic. There may be many reasons for this non-determinism. The software may rely on third-party components or services with uncertain performance; the communication channels between distributed components may be of variable quality; the components may be coupled to an uncertain physical world; a mixed-initiative system may have unreliable or variable humans in the loop; and so on. Just as contracts help provide predictable behaviour in the social world, software contracts can be used to regulate the associations, and thus the behaviour, in loosely-coupled systems.

This chapter shows how contracts can be used to create, monitor and regulate the interactions between roles in a ROAD organisation. Section 6.1 briefly reviews the various uses of the concept *contract* in software engineering. In Section 6.2 we define the properties of a ROAD contract, and then, in Section 6.3, show how the control of interaction in a contract can be abstracted from its functional properties. In Section 6.4 we show how performance measurement points that correspond to various synchronisation approaches can also be defined at this abstract level. Section 6.5 elaborates the general properties of application specific concrete contracts that inherit from these abstract contracts. Section 6.6 relates the concept of a service-level-agreement to ROAD contracts. This meta-model of contracts forms the basis for our implementation of contracts in the ROAD framework, described in Part III of the thesis.

6.1. Software contracts

The concept of a *contract* is commonly used in software engineering, although a variety of meanings have been attached to the term. For example, Bertrand Meyer's design-by-contract (DBC) (Meyer, 1988) defines the preconditions, post-conditions and invariants that must hold for a given type of interaction with an object. Such contracts are essentially one-sided because they only explicitly express the conditions for one party. The other party is anonymous. DBC contracts are a type of interface enforcement.

In the real world, however, contracts always have at least two parties. They are a type of association that expresses the obligations and responsibilities the parties have to each other. This view of a contract as a multi-party association has also been proposed in software design. Richard Helm (Helm, Holland and Gangopadhyay, 1990) saw contracts as a way of specifying behavioural compositions and the obligations of participating objects. This concept can be seen as a precursor to the reusable design patterns which Helm went on to develop with the Gang-of-Four (Gamma, Vlissides, Johnson, and Helm, 1995). Although Helm envisaged the development of programming language constructs to instantiate contracts that captured such interactional compositions, such patterns have remained largely design artefacts. More recently, however, just as DBC contracts have been implemented with aspects (Diotalevi, 2004), work has been done to encapsulate such patterns with aspects (Hannemann and Kiczales, 2002).

In Chapter 3 we reviewed, what we termed, *contract-oriented* architectural frameworks (Collet, Rousseau, Coupaye et al., 2005; Mukhija and Glinz, 2003). These frameworks use contracts to compose configurations and/or to constrain interaction between components. The ConFract framework (Collet, Rousseau, Coupaye et al., 2005) we reviewed in Chapter 3 is based on Beugnard et al.'s (1999) classification of contracts. This classification proposes four "levels" of contract.

- Level 1: Syntactic contracts - ensure compatible signatures for interaction.
- Level 2: Behavioral contracts - define a component-centric interface similar to DBC pre, post and invariant conditions
- Level 3: Synchronization contracts - deal with concurrency/coordination issues across multiple components.

- Level 4: Quality of Service (QoS) contracts - encompass all non-functional requirements and guarantees.

However Beugnard, and consequently ConFract, sees these levels/contracts as separate contracts rather than the terms of a single contract as in ROAD.

In ROAD, contracts are used not only to compose and control associations between roles, but are also used to make role-players 'accountable' for their performance. In other words, ROAD contracts not only define functional relationships, but also define the non-functional properties of those relationships, both in terms of the requirements (obligations of the parties) and the state-of-fulfilment of those obligations (performance). In this sense they are more like a commercial contract than DBC contracts or contracts that only express behavioural compositions. Such performance contracts are a way of monitoring and controlling the associations between entities that are loosely coupled. They specify the required performance, and monitor and store the actual (measured) performance of a role-player in an organisation. To extend the analogy with a commercial contract, ROAD contracts are more like employment contracts than, say, a building contract where the contract specifies that a set of tasks be completed once-only. ROAD contracts empower the role-player in the context of the organisation, and set the expected performance levels for repeated actions.

In brief, ROAD contracts combine all the three aspects discussed above: composition, interaction control, and performance.

6.2. The attributes of ROAD contracts

ROAD contracts are binary association classes that express the obligations of the contract parties to each other. They are both a specification of the parties' mutual obligations, and a runtime entity that monitors and, to some extent, enforces compliance with that specification. ROAD contracts define the functional interactions that can occur between the role players, define the non-functional requirements of each of the parties with respect to those interactions, and measure the role-players' performances against those requirements.

ROAD contracts have the following features (illustrated in Figure 6-3 below):

- The names of the parties to the contract. A ROAD contract binds roles of particular types (e.g. Foreman, ThingyMaker).

Contracts will also have a number of clauses. Clauses can be of three types: terms; general clauses; and protocol clauses:

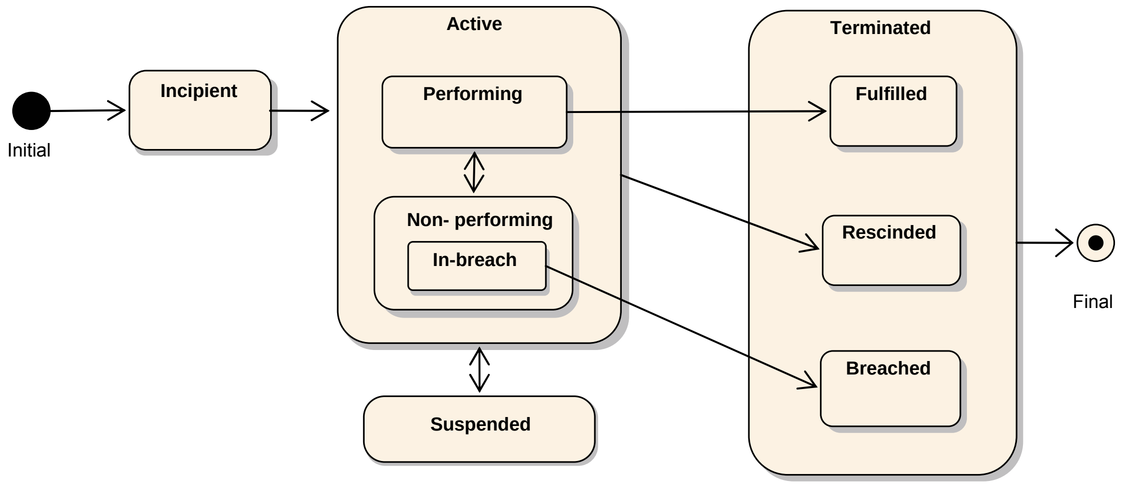
- The *terms* of a contract are clauses that specify what one party can ask of the other party. The collection of terms defines the parties' mutual functional obligations. For example, a contract term may specify that a ThingyMaker is obliged to fulfil requests to make thingies from its Foreman. Non-functional attributes (utilities) are associated with those terms – for example, the minimum performance standard, the price, quality of service etc. Each term of the contract can have one or more agreed utility functions that define how performance of actions taken under that term of the contract will be measured. Contracts may also contain provisions that define remedies if a clause is breached, or if there is underperformance, by one of the parties. Some terms may "go to the heart of the contract" in which case breach of a clause leads to termination of the contract.
- *General clauses* in a contract define the preconditions for the contract's instantiation. These include any conditions relating to commencement, continuation, and termination of the contract. In ROAD, contract *termination* is the cessation of an association (similar to termination of employment) rather than the completion of a task.
- *Protocol clauses* define sequences of terms to be followed by the parties (Yellin and Strom, 1997; Plasil and Visnovsky, 2002). For example, a foreman might be required to allocate the resources (thingy parts) to a ThingyMaker, before it can ask the ThingyMaker to make a thingy. A number of existing approaches exist for specifying interaction protocols (Bracciali, Brogi and Canal, 2002). Examples of protocol clauses can be found in the Buyer-Seller contract described in Chapter 10.

As well as having the above attributes, ROAD contracts have a variety of manifestations: *general form* (*à la* class), *specific contract* (*à la* object) and an *execution state*.

- The *general form* (type) of a contract sets out the mutual obligations and interactions between parties of particular classes (e.g. Foreman, ThingyMaker). Clauses applicable to all contracts of that type can be defined. Such clauses may express interaction patterns (Party A will do α under term X when Party B has done β under term Y). Clauses may themselves be the subject of interaction patterns (Clause 2 will take effect after the state defined in Clause 1 is fulfilled).
- A *specific* contract puts values against the variables in the contract *schedule* (e.g. Foreman and ThingyMaker are named, date of commencement agreed, performance conditions put on clauses etc.). Extra clauses, not in the general form of the

contract, may also be added. A specific contract is *instantiated* with an identity when the concrete contract is 'signed'; that is, when specific parties are bound to the contract.

- The terms of a contract have *execution states*. Each term has a state machine that maintains the state of interaction between the parties with respect to that term (e.g. Party A has asked Party B to do α under the terms of Term X, but Party B has not yet complied). We call the interaction that occurs during the execution of a term, a *transaction*. States of an instantiated contract can include incipient, active, suspended, and terminated. Active and terminated states can have a number of sub-states as shown in the diagram of a contract life-cycle[1] in Figure 6-1 below.



.

Figure 6-1: Life-cycle of a contract showing various states

The transitions between these contract states are instigated by the contract's owner (the organiser of its self-managed composite), or by clauses being triggered. The triggering of conditions in contract *terms* can cause transitions between the Active state's sub-states. For example, Figure 6-2 is an example of a (partial) Foreman-ThingyMaker Contract. The specific schedule values of the contract instance are in italics. If the contract state is Performing, and Term 1 is violated (the ThingyMaker on average makes less than 5 thingies per minute), then the contract state will transition to Non-performing or In-breach (if the rate is less than 2 thingies per minute). If there are no general clauses that match this circumstance, the Organiser needs to make a decision as to whether it

[1] (Beugnard, Jézéquel, Plouzeau et al., 1999) defines a similar scheme for contract life-cycle, namely: Definition (selection of contract type as suitable), subscription (parties bound to contract), application (including handling contract violations), termination (parties unbound), and deletion. In ROAD, on the other hand, parties (role instances) are bound to a contract when it is instantiated.

will transition the contract to one of the Terminated states that ends the association between the roles. This is not necessarily an automatic transition, as the Organiser may have no other thingyMaker players available and may choose to keep an underperforming player. Note that contract states do not explicitly indicate the "guilty party", but this information is implicit in the term that is violated (Party B in the case of Term 1).

Contract Type Foreman-ThingyMaker Contract	
Contract Instance Name	*ft1*
Parties Party A: Foreman	*f1*
Party B: ThingyMaker	*tm1*
Terms	
1. B *must* make thingies on request from A	*NFR1 Moving average >= 5 thingies / minute*
PRE: qty (thingyparts) > 0	*NFR2: cost = $1 / thingy*
2. A *may* provide thingy parts to B	
on request from B	*NFR: none*
Breach conditions	*B provides < 2 thingies / minute*
General Clauses	*contract state is* suspended *if B under maintenance*
Current state	*Incipient*

Figure 6-2: Example of a contract instance between a Foreman and a ThingyMaker

Figure 6-3 below shows the basic concepts that make a ROAD contract, and that serve as a meta-model for the implementation of contracts in the ROAD framework as described in Part III. The elements in the diagram are discussed in more detail in subsequent sections of this chapter.

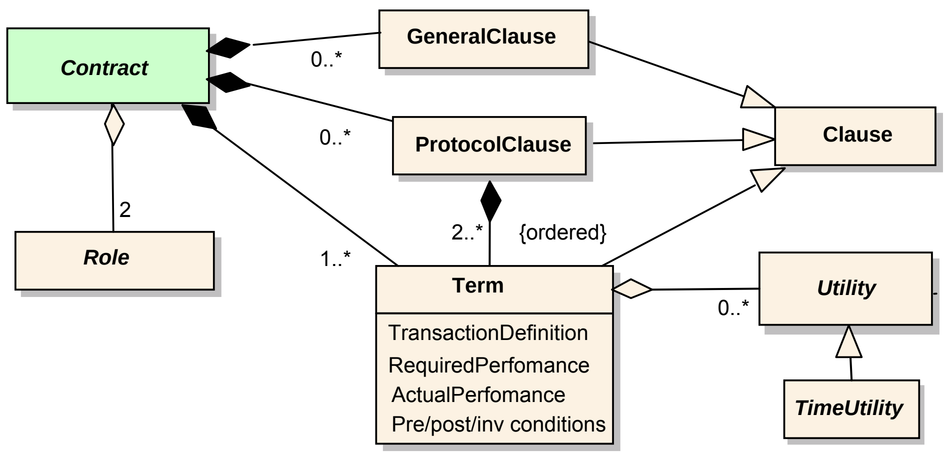


Figure 6-3: ROAD Contracts - Basic Concepts

Real-world commercial contracts are passive bits of paper that are monitored, and to some extent enforced, by the parties. A ROAD software contract, on the other hand, can store dynamic state-of-execution information in the contract itself. The contract itself (on behalf of the organisation) can also enforce the terms of the contract by controlling the interactions between the parties.

In ROAD, contracts between roles are currently binary. While there is no technical impediment to creating contracts with more than two parties, if a number of parties are jointly responsible for the performance of a contract term it can be difficult to assign responsibility in the event of failure. As one purpose of ROAD contracts is to identify underperforming role players, making all contracts binary simplifies this process. The limitation of binary contracts is that they cannot, by themselves, model complex interdependencies between three or more parties. In ROAD, these interdependencies between contracts are handled by the composite's organiser as we describe in the next chapter.

6.3. Contract abstraction

Contracts between functional roles often share common characteristics with respect to the *control* aspects of the relationship. These aspects can be abstracted and encapsulated as abstract contracts. We call these abstract associations *performative contracts* because they express what one party can ask of another; i.e. a performative[2]. The participants in a performative contract play performative roles, as well as functional roles. Examples of such performative roles include supervisor, subordinate, auditor, auditee, predecessor, successor, buyer, seller, and so on. As such, performative roles are classifiers for the ends of an association, and always come in pairs as shown in Figure 6-4 below.

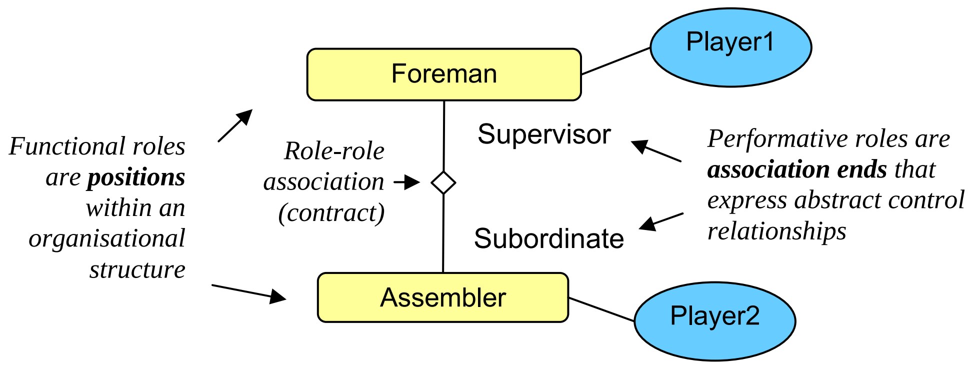

Foreman
Player1
Functional roles are **positions** within an organisational structure
Role-role association (contract)
Supervisor
Performative roles are **association ends** that express abstract control relationships
Subordinate
Assembler
Player2

Figure 6-4: Functional and performative roles

Using our example of a WidgetDepartment, the performative relationship between a Foreman and an Assembler could be characterised as a Supervisor-Subordinate relationship. Similarly, a Foreman-ThingyMaker relationship would also be characterised as a Supervisor-Subordinate relationship. Rules can be defined that control the interactions between such *performative* roles. For example, a supervisor can tell a

[2] In previous work (e.g. (Colman and Han, 2006b)) we called these *operational-management* (or just *management*) roles and contracts. In order to avoid confusion with composite management (organiser) roles we now call them *performative* roles and contracts. We have adopted the word *performative* from

subordinate to do a work related task but a subordinate *cannot* tell a supervisor what to do. At the programming level, this means that subordinate roles cannot invoke particular categories of methods in supervisor roles. Performative contracts define patterns of interaction between roles at an abstract level but, unlike *functional* roles, do *not* serve as a *position* in an organisational structure. Abstracting away control from the process of an organisation facilitates the reuse of common patterns of control across various types of contract, and can be used to describe the global flow of control through an organisation. For example, an organisation that is a network of roles bound by supervisor-subordinate performative contracts could be characterised a command hierarchy.

Concrete contracts inherit control relationships from these performative contracts. The conceptual relationships between concrete and abstract performative contracts, and the respective roles they associate, are illustrated in Figure 6-5 below.

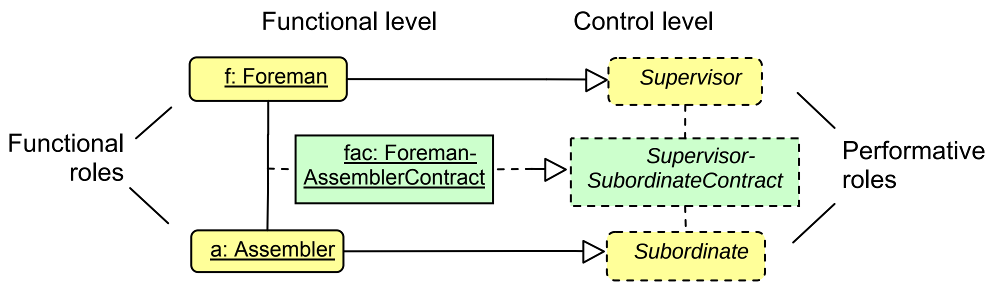


Figure 6-5: Functional and performative contracts

As can been seen from Figure 6-5 above, fac is an *instance* of a Foreman-AssemblerContract which is an association class between the Foreman and the Assembler roles. The ForemanAssemblerContract inherits the form of its control relationships from the SupervisorSubordinateContract by virtue of the fact that the Foreman has a Supervisor role in relation to the Assembler's Subordinate role.

Figure 6-6 below illustrates an organisational structure for our WidgetDepartment. This structure is similar to the Bureaucracy pattern (Riehle, 1998) but is built using contracts. In order to simplify the diagram, contracts have been drawn as diamonds. The structure, which is similar to a business organisation chart, is still abstract because players have not yet been assigned to roles. Note that the DooverMaker *functional* role has *both* Peer and Subordinate performative roles in the organisational structure relative to other functional roles with which it interacts.

agent communication languages to indicate, at an abstract level, what one party can say to another (i.e. what types of speech acts are permissible).

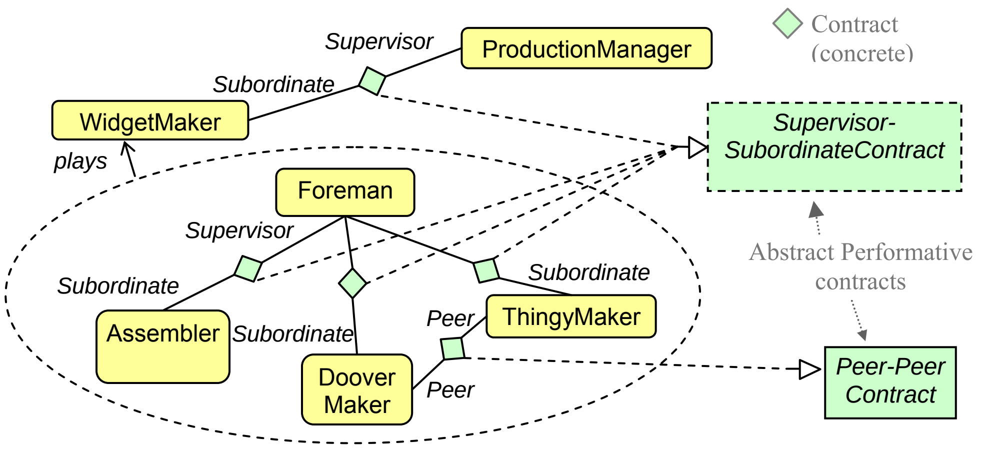


Figure 6-6: Domain specific abstract organisation bound by contracts

6.4. Abstract contracts at the performative level

A contract *term* defines a transaction that expresses the obligation of one party to the other party. As discussed earlier, the communications that occur between the parties in such a transaction can be viewed, at an abstract level, as expressing control relationships. Communications between the parties can be represented as abstract control messages. This allows us to characterise three properties of contracts and terms: the authority relationship between parties to the contract; the sequence of abstract messages in a term; and the *points* at which the performance (or non-performance) of the contract term can be measured for various types of transaction. These three properties of abstract performative contracts are examined in the next three subsections.

6.4.1. Expressing authority using abstract message types

In ROAD, all application-related interactions between players occur via the contracts that associate their roles in the organisational structure. This control over interaction can become important if the players of those roles are, say, outside the organisational boundary and their behaviour is uncertain. Such players will become increasingly common as software systems become more open, dynamic and complex. Contracts *restrict* the types of method that particular role instances can invoke in each other, thus helping to ensure that role-players are well-behaved with respect to the application. While such restrictions can always be written into domain-specific concrete contracts, there are a number of common patterns of authority relationship that can be generalised into abstract performative contracts. From our example above, the Supervisor-Subordinate performative role association restricts interactions between the entities playing the ThingyMaker and the Foreman to certain *types* of interaction. For example, a ThingyMaker cannot invoke actions in a Foreman-Supervisor. Furthermore, these contracts only *allow* interactions between particular instances of role-players. For example, the method ThingyMaker.setProductionTarget() can only be invoked by the ThingyMaker's *own*

Foreman. A Foreman that is not contracted to that particular ThingyMaker could not invoke the method.

The control communication in performative contracts can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the *control* aspects of the communication from the message. We can define a simple set of abstract CCAs for direct and indirect (resource constraint) control and for information passing (Colman and Han, 2005). Table 6-1 defines an example set of such abstract message primitives suitable to a hierarchical organisation.

Table 6-1: Example set of abstract message types represented by CCA Primitives

D	DO	Q	QUERY	R	RESOURCE_REQUEST
G	SETGOAL	I	INFORM	A	RESOURCE_ALLOCATE
Y	ACCEPT	N	REJECT		

Expanding our example of a Supervisor-Subordinate contract using the primitives we defined above, Supervisors can initiate some types of interaction and Subordinates can initiate others. Initial CCAs for these roles are:

Supervisor initiated:	DO, SET_GOAL, INFORM, QUERY, RESOURCE_ALLOCATE
Subordinate initiated:	INFORM, QUERY, RESOURCE_REQUEST

Other forms of performative contract, such as Peer-Peer, would have different sets of valid initial CCAs for each party. For example, peers might be able to initiate messages corresponding to all the above CCA types, but the respondent peer has the option of replying with a REJECT. We discuss sequences of CCAs in the following section.

If needed, the set in Table 6-1 could be extended to capture a referential command relationship (A tells B to tell C to do something). Alternatively, the set could be expanded to express propose-commit type communications that might be found in agent communication, or in a database two-phase commit. However, while the above set is only a basic set of CCAs, it is sufficient to allow us to define a number of contracts between performative roles. From these contracts we can create organisational structures such as those in Figure 6-6 above.

The concept of a CCA in this paper is similar to the concept of a *communication act* (or performative) in agent communication languages such as FIPA-ACL (The Foundation for Intelligent Physical Agents, 2002). CCAs, as defined here, are far more limited in their extent. CCAs deal only with control communication of two parties bound in an organisational contract, and do not have to take the intentionality of independent agents into account (Zambonelli, Jennings and Wooldridge, 2003). Abstract communication act types have also been used to control interaction in Web services Message Exchange Patterns (MEPs) in WSDL (W3C, 2005). However, MEPs only express the direction of communication (IN or OUT) and whether or not the communication is robust or optional. In (Barros, Dumas and ter Hofstede, 2005) a number of service interaction patterns are catalogued, including both bilateral and multilateral patterns. The bilateral patterns (e.g. “Relayed Request”) may provide a catalogue from which transaction types expressed as CCA sequences could be developed. Abstract protocols also appear in component composition (Yellin and Strom, 1997). These are aimed at ensuring the compatibility of component interfaces and the sequencing of messages. They express the direction of the message and if it is a request or a response (Plasil and Visnovsky, 2002). Authority relationships or control semantics (eg. DO) are not captured.

6.4.2. Contract transactions as CCA sequences

As well as controlling individual method invocations, contracted transactions often involve a sequence of interactions. As shown in Figure 6-3 above, a contract term is associated with a transaction definition (as distinct from a protocol clause which is a sequence of transactions). The contract needs to track these interactions to ensure that contractual obligations are being followed by the parties, and to know when a contractual transaction is completed. A transaction instance performed under a term of the contract can be viewed as a sequence of CCAs. These CCAs are abstractions of the actual underlying messages or method invocations. These sequences are at the same level of abstraction as CCAs. Only the form of communication between the parties is represented. There is no information about the particular content of the task. There are a limited number of these sequences that form sensible interactions. For example, both QUERY $\rightarrow$ INFORM, and RESOURCE_REQUEST $\rightarrow$ RESOURCE_ALLOCATE make sensible CCA sequences between an initiator and respondent, whereas DO $\rightarrow$ RESOURCE_ALLOCATE (presumably) does not make sense.

What constitutes a valid sequence also depends on the type of request synchronisation used in the transaction. The CORBA middleware standard defines four approaches to request synchronisation: *Oneway*, *Synchronous*, *Asynchronous* and *Deferred-Synchronous* (Emmerich, 2000). Similar distinctions between synchronisation approaches are needed in the definition of valid CCA sequences. For example, a oneway or synchronous request does not require any separate response from the respondent. In these circumstances a single DO type invocation may be a valid transaction. On the other hand, if an asynchronous approach is implemented a reply would be expected in the form of, for example, a DO-INFORM sequence.

It is possible to represent these sequences as regular expressions made up of CCAs between initiator and respondent. To do this we will encode the CCAs as single letters (as in Table 6-1 above) so that complete transactions can be represented as individual strings. For further expressiveness, we can apply the convention that initiator CCAs are capitalised, and respondent CCAs are in lower case. For example, the string "Dy" indicates that the initiating party asks the respondent to do something, and that the respondent subsequently accepts. A deferred-synchronous DO sequence (where the initiator is responsible for getting the result of the transaction) could be expressed as "DQ". Other sequences, such as a deferred-asynchronous ("DyQi"), could also be defined if needed.

Table 6-2: Example form of an abstract performative contract

Performative Contract	
Name	Supervisor-Subordinate
Party A	Supervisor
Party B	Subordinate
A initiated terms	
- Oneway	I; G; A
- Synchronous	D; Q
- Asynchronous	Dy; Di; Gy; Gi; Qi
- Deferred Synchronous	DQ, GQ
B initiated terms	
- Oneway	I
- Synchronous	Q; R
- Aynchronous	Qi; Qn ; Ra; Rn ; Ri

The form of our Supervisor-Subordinate performative contract has been summarised in Table 6-2. The valid invocation sequences in a transaction are expressed as strings as defined above. The contract defines terms that permit asynchronous and deferred-synchronous interaction. In the example contract, Oneway DOs are not permitted as we want to measure the time-performance of action requests, and to do this we need a response when the transaction is complete. The terms of the contract will be violated if the sequence of interactions does not follow one of these strings. Other types of performative contract will have different sets of permissible sequences. For example, the sequence "DiN" is presumably acceptable in a Peer-Peer performative contract, where the "i" is a conditional accept and the initiator rejects the condition. A term of a concrete contract, such as shown above in Figure 6-2, would be characterised by one of the above CCA sequences. For example, Term 1 in "B *must* make thingies on request from A" might be a "Di" sequence, assuming the transaction was asynchronous.

We can further extend the CCA sequences to take account of a non-response to a message where, say, communication channels are unreliable. For example, D*xi would express the situation where the initiator can send a DO type message up to x times before there is a violation of the contract term. For each transaction where a response is expected, values for the response timeout and the number of permissible retries (x) would be specified (this is done at the concrete level of the contract).

If contracts are to enforce CCA sequences, they need to keep track of the *state* of communication between the roles that are party to the contract. This implies that there must be an instance of a contract for every association between roles. For machine processing, control abstractions of transactions can be represented by finite state machines (FSMs). These FSMs keep track of the transactions between two contracted parties and report violations. Clauses can have as a goal the *maintenance* of a state or the *achievement* of a state. In the case of maintenance terms, a successful transaction will result in a return to a 'ready' state. The successful completion of achievement clauses will result in a 'completed' state for that clause. Figure 6-7 below illustrates DO transaction sequences for the Supervisor-Subordinate contract. The nodes represent CCAs issued by either the initiator or the respondent in the transaction of a particular term in the contract. The letters in the nodes are a short hand for CCAs, as defined Table 6-1 above (initiator CCAs in capitals, respondent CCAs in lower case). The FSM shows *valid* synchronous, asynchronous and deferred-synchronous transactions initiated with a Supervisor DO (for simplicity only timeouts on the DO has been illustrated).

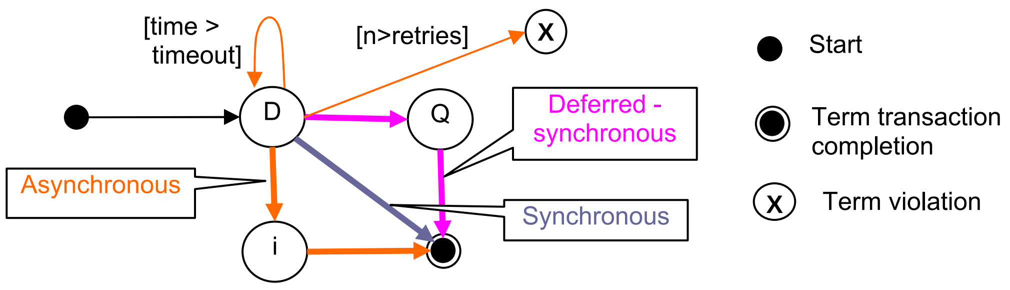


Figure 6-7: Valid synchronous, asynchronous and deferred-synchronous CCA sequences initiated with a Supervisor DO CCA

6.4.3. Using CCAs to define performance measurement points

In an adaptive system, we need to know if the role-players are performing their role(s) according to the non-functional requirements (NFRs) defined in the role's contracts, so that adaptive action can be taken if underperformance is detected. The NFRs are defined *with respect to* functional transactions (e.g. *Functional transaction* Task A will be completed in *NFR* Time t). CCAs allow us to define transaction patterns in a domain-independent way. The performance, or non-performance, of an obligated party bound by a contract term can be determined by measuring the change in various states at the start and at the end of the transaction. By intercepting messages between roles, and monitoring the CCAs of those messages, a performative contract can determine the start and finish of transaction patterns defined by contract terms. The change of state measured can be of two types: time-dependent and domain-state-dependent. Time-dependent performance is calculated by measuring the time it takes to perform the transaction itself. Calculating domain-state-dependent performance involves measuring some state of the domain or environment before and after the transaction, and then determining the effect of the transaction (e.g. a cost-function is evaluated, or a control–variable is measured.) In ROAD, performance is measured by an arbitrary utility function associated with the contract-term, as shown in Figure 6-3 above.

The type of performance that can be measured, and how CCAs indicate the start and finish of the transaction, will depend on the synchronisation method of the transaction; that is, whether the contract term is oneway, synchronous, asynchronous or deferred synchronous.

Timed performance may need to be measured when the term invokes some action in the obligated party; that is, when a DO or SETGOAL type message is sent. Figure 6-8 below shows the interception points at which performance is measured for the differing synchronisation methods. In synchronous interaction, time is sampled at the point of method invocation and at the point of return of the method. Asynchronous interaction relies on detecting the INF CCA reply that the obligated party sends when the task is finished. In the case of a deferred-synchronous interaction, the execution time cannot be measured directly from interactions intercepted by the contract between roles. This is because there is no message sent on completion of the task from the obligated role to the invoking role. However, the performance of this type of interaction can be measured against a benchmark: the time between the invocation and when the query for results is sent from the invoking role. The obligated party either meets the benchmark or does not.

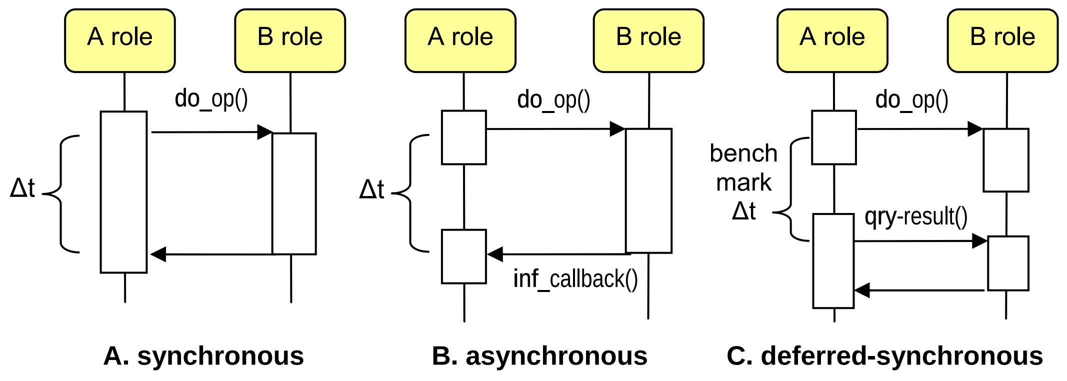


Figure 6-8: CCA points intercepted for measurement of time-based performance vary depending on the contract-term's synchronisation type (players not shown)

Figure 6-8 above is a simplification; it does not show the role-players to whom the calls are delegated, or show the contract that intercepts the messages between the roles. It is this contract that measures the time at the appropriate points, and calculates the time-based metric. Also note that performance, as measured between abstract messages, does not indicate success or failure of the task itself, as there is no domain semantics expressed. All that is measured is the time elapsed. This measurement needs to be interpreted by the contract into a level of performance.

Domain-state-dependent performance measures the change of state in environment rather than time elapsed during the transaction. By detecting the CCAs with which a transaction starts and finishes, a contract can be made to evaluate an arbitrary utility function that indicates some change of state within the domain or environment (including changes of state in the player). This utility is defined at the concrete level of the contract. Such measurement of performance can be applied to transactions with all types of synchronisation method. Even a oneway interaction, where no response is received from the obligated party, can be monitored for domain performance provided there is an appropriate delay between the measurement of the initial state and the measurement of the altered state. For example, a classic feed-back control loop (as described in Chapter 2) is a oneway interaction. The process set by the controller does not return a result – rather a control variable is sampled in the environment to determine the result of the control settings. Figure 6-9 below shows the measurement of domain-state dependent performance in an asynchronous transaction. Role A invokes some action in Role B. This invocation is intercepted by the contract which measures the environment ε before passing invocation to Role B, resulting in the invocation being enacted by B's player. This action changes the environment.

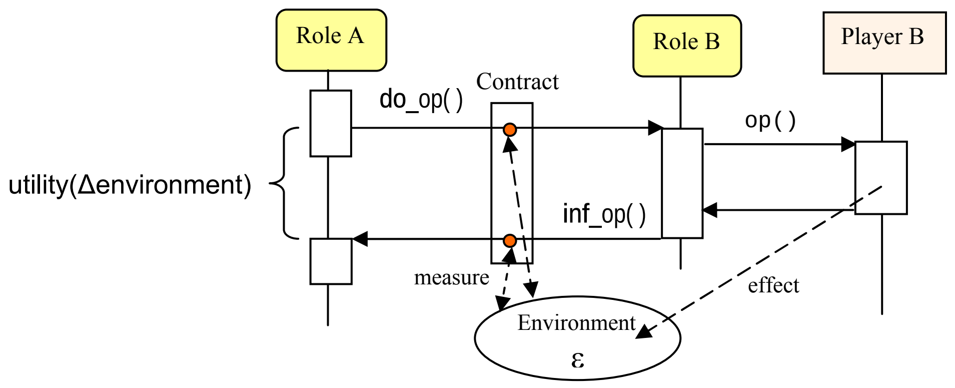


Figure 6-9: Change of domain-state measured in an asynchronous transaction

The response message is intercepted by the contract and any change in the environment ε is measured. For example, if Player B is an external service that charges for the provision of a function, the contract could access the accounting system before and after the transaction.

Contracts at the performative level are limited in that they only monitor or enforce the *form* of the communication between the parties. Abstract types of interaction may be restricted, and transaction patterns monitored for performance — but there is no domain content apparent at the performative control level of abstraction. Domain-specific content is defined at the concrete, functional level of the contract.

6.5. Concrete contracts

A concrete contract type defines the types of functional role that can participate in the contract. In addition, as was illustrated in our example of a Foreman-ThingyMaker contract in Figure 6-2 above, an instance of a concrete contract further *specifies* the terms and clauses of the contract. The specification of a term includes the specification of the signatures of the invocations involved in the transaction. The term can also define an expected level of performance associated with its transaction, as measured by an associated utility function. A concrete contract inherits its control patterns from the abstract performative contract as described above. For example, the Foreman-ThingyMaker contract inherits the CCA control patterns from the Supervisor-Subordinate abstract performative contract, which are applied to the specific methods of the Foreman and ThingyMaker (see below).

All contracts need to specify the following items:

- ***Parties***. The *types* of functional role that can be bound to the contract are specified. In the example contract in Figure 6-2 , only instances of roles of type Foreman and type ThingyMaker can be bound to the contract.
- ***Terms*** of the contract. Each term defines a transactional obligation of one party to the other. These are expressed as an initial method signature that can be invoked in the obligated party. When a contract specialises an abstract performative contract, all functional role method invocations and responses between the parties are associated with CCA primitives. For example, the `do_makeThingy()` method of the ThingyMaker tm would be matched to the DO CCA. This means a supervisor, contracted to tm, can invoke this method. If a CCA sequence is to be enforced for a transaction, a valid CCA *regular expression* (as defined in the performative contract) is assigned to the interaction. Transactions between the parties must follow any abstract CCA pattern defined in the performative level of the contract. If a CCA sequence allows for timeouts, the values for timeouts (in the event of no response), and values for the number

of retries that are permitted, are specified. This is done in the concrete contract as these values only make practical sense in relation to a domain specific function.

Optionally, contracts may specify performance:

- ***Performance of terms***. The contract defines the required level of performance and measures actual performance. If a term defines a type of action that can/must be performed, the contract term can have associated with it a utility function that measures the obligated party's actual performance of those actions. In the example contract (Figure 6-2), thingies must be made at a rate > 5 per minute. As set out in the previous chapter, these utility functions can be time-dependent or domain-state dependent. The actual performance is compared to the required performance to determine the obligated party's level of performance with respect to the term – e.g. performing, underperforming, in-breach. For example, a contract might specify the maximum time allowed for a contracted thingyMaker to produce a thingy. These non-functional requirements (NFRs) reside in, and are measured by, the contract rather than the component itself. As such, the performance requirement of a term can be changed dynamically by the composite organiser.
- ***Performance of contracted party***. The purpose of measuring the performance of contracted parties is to attempt to mitigate underperformance, and to replace an underperforming party if necessary. A contract term always has one party that is responsible for its performance. The performance of a party, with respect to the contract, is the aggregation of its performance of such terms. The significance of underperformance can vary. Some term violations "go to the heart" of the contract and violation of the critical clause leads to automatic breach of the contract — the contract throws an exception. On the other hand, other terms may not be as critical to the contract and the underperformance will merely be recorded. The contract contains metrics to measure the performance of its terms; for example, 'average time to make a thingy'.

Additionally, conditions for general clauses and terms may be specified:

- ***Preconditions, post-conditions and invariants*** for the interactions can be specified for both general-clauses and terms. General clauses that set preconditions for the contract's instantiation can be defined. These include conditions relating to commencement, continuation, and termination of the contract. In our example, the contract is suspended if the ThingyMaker machine is off-line due to maintenance. Conditions can also be set against the performance of contract terms. These conditions are similar to those defined in the design-by-contract (DBC) approach (Meyer, 1988), where such conditions are aimed at ensuring the correct functioning of the software. Consequently, these DBC conditions themselves cannot be changed. In addition to such fixed conditions, ROAD contracts may have conditions that *can* be varied by the organiser (provided the variation does not contravene correctness constraints). This allows variable NFRs to be expressed as conditions of the contract. For example, if there are costs associated with the performance of a function, such as making a thingy, then the contract might specify the acceptable limit of those costs.

During execution, a contract itself monitors the interactions between the roles. The contract will prevent unauthorised or invalid interactions and monitor transactions in order to maintain the state of execution of its clauses. The contract also keeps the state of any performance metrics updated. If an obligated party underperforms according to a term of the contract, or if a clause is violated, the contract informs the organiser role that controls it. Contracts may also be actively monitored by their organiser roles. We examine the interactions between contracts and organiser roles in the next chapter.

6.6. ROAD contracts and role-player bindings

It is important to remember that ROAD contracts are always *internal* to the organisation. They define the relationships between roles, and roles are always a function defined by the organisation. The player of a role can, however, be *outside* the organisation boundary. For example, the doovers for our Widget Making department are sourced from an external service. The relationship between ROAD contracts and the role-player bindings discussed in the previous chapter therefore need to be considered.

In the discussion of functional roles in the previous chapter, we pointed out that a role description is an aggregation of the properties derived from its relationships. To be able to play a role, a player must be able to meet the requirements defined in the role. It follows, given the above discussion of contracts, that the properties of a role-player binding are the aggregation of the obligations of the role expressed in the role's contracts. This role-player binding may itself be a contract of sorts that is negotiated between the organisation (as owner of the role) and the player. We will call these role-player contracts Service Level Agreements (SLAs) in order to distinguish them from binary ROAD contracts. SLAs can be external to the organisation boundary, and their formulation may therefore be a matter of negotiation between organisational entities, rather than derived solely from organisational fiat.

While we do not discuss the format of role-player bindings or SLAs in this thesis, it is worth noting that SLAs (or more correctly the organisation's negotiating position for what it wants in a SLA) can be derived from the role's contract terms. In a Web service context, ROAD contracts provide a natural mapping to coordination mechanisms such as WSLA (IBM Corporation, 2003), WS-Coordination (BEA Systems, IBM and Microsoft, 2004), or external contracts such as WS-Agreement (Global Grid Forum, 2004). For example, an internal contract that binds the role (as proxy for the Web service in the organisation) to the rest of the system can serve as a WS-Agreement *agreement template*, as shown in Figure 6-10 below. ROAD contract terms map to *service description terms* and *service level objectives* in WS-Agreement. ROAD contracts have no concept of a *guarantee term* as found in WS-Agreement because there is no notion of *penalty for default* in an internal ROAD contract. Such terms could however be added to the template when the external contract is defined.

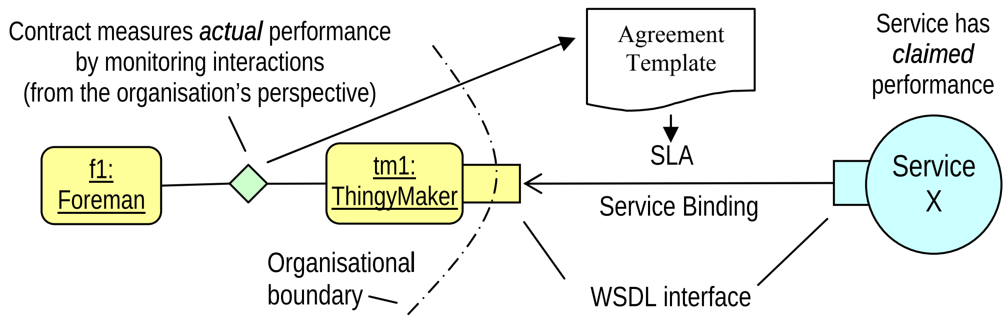

Contract measures *actual* performance by monitoring interactions (from the organisation's perspective)
Agreement Template
Service has *claimed* performance
f1: Foreman
tm1: ThingyMaker
SLA
Service X
Service Binding
Organisational boundary
WSDL interface

Figure 6-10: Internal ROAD contract maps to external SLAs

6.7. Summary

Contracts in the ROAD framework provide the connectors between roles that create the organisational structure. ROAD contracts serve three functions in an adaptive system: composition, interaction control, and measurement of performance over interactions. ROAD contracts have abstract performative and concrete functional levels. Abstract performative contracts specify the type of communication acts that are permissible between the two parties, and they define the transaction patterns that can be measured for performance. Concrete contracts bind the role instances to the contract, and create the clauses of the contract that specify, among other things, the performance obligations of each party. Abstracting the performative aspects of contracts makes possible, through contract inheritance, the reuse of authority patterns found in many types of organisational structure.

In the next chapter, we show how roles and contracts are combined together to create self-managed composites, and how these self-managed composites are structured to create adaptive applications.

7

Self-managed Composites and the Management System

A self-managed composite (“composite” for short) is a cluster of role instances bound by contracts. Each composite has a single organiser that manages its internal roles and contracts. Each composite serves some definable function and, as such, is itself a role-player that plays a role in an ‘enclosing’ composite. Self-managed composites are the adaptive unit in a ROAD application. Such composites are “self-managed” because they attempt to adapt to meet the dynamic requirements defined in the external roles they play, and also adapt to the variable performance of the players that play the internal roles of the composite. In terms of the system-theoretic concepts discussed in Chapter 2, self-managed composites are systems with a well-defined boundary that attempt to maintain a homeostatic relationship between their external role and their internal components. A ROAD application[1] is a network of self-managed composites which contain roles. These roles can be played by other self-managed composites or other players.

In this chapter we begin by describing the structure of self-managed composites and their relationship to the other elements in the ROAD framework: roles, players, contracts and organisers. We then discuss the issues of message delegation, player containment and whether or not composites perform any domain process or maintain state. Section 7.2 describes the function of the composite organiser who is responsible for managing the composite. The functions of an organiser *role* are distinguished from

[1] The top level player is the application itself. While the application does not in itself play a role in an enclosing ROAD composite, the application can model itself as a role player in its environment. The description of this environmental model, and the application as a role within that model, is a promising subject for further research, but is beyond the scope of this thesis.

that of an organiser *player*. Organiser roles define *how* to change the composite while players *decide* what to change. An example strategy for adaptation *within* a self-managed composite is then described. Section 7.3 describes the management system which connects the organisers *across* composites. This management system is distinct from the functional system of functional roles and players. Adaptive behaviour *across* composites is described including the propagation of non-functional requirements and performance information. The conclusion to the chapter includes a summary model of all the main ROAD concepts presented in this part of the thesis (Chapters 4 – 7).

7.1. The structure of self-managed composites

The relationship between roles, players, contracts and composites is illustrated in Figure 7-1 below. A self-managed composite consists of one or more contracts, each contract associating two functional roles (FnRole). Each role is played by a player, which may itself be, recursively, a composite. This recursive structure of self-managed composites is the basis of distributed management in ROAD applications. Each composite has one organiser (role and player) which manages the indirection within the composite. The elements in the conceptual model in Figure 7-1 form the basis of the ROAD framework, and can be viewed as abstract classes from which concrete domain-specific classes are specialised.

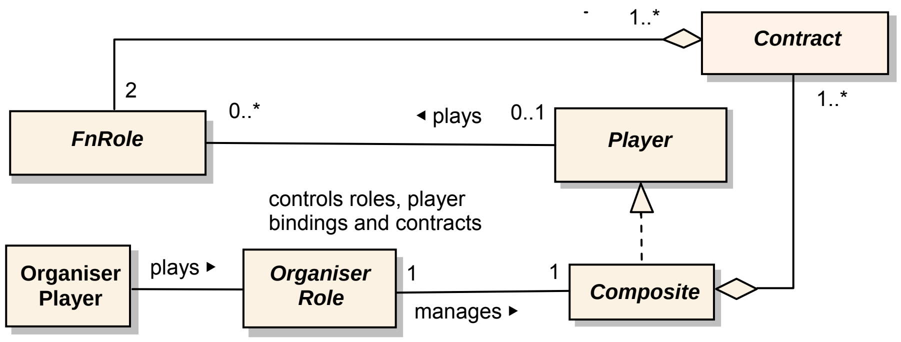


Figure 7-1: Conceptual relationship between functional roles, players, and self-managed composites

As well as being a container that aggregates contracted roles, a composite-player must present a functional interface(s) that is compatible with the external role(s) it plays. (As a player can play more than one role, it may present a number of such interfaces). It follows that, to be well-formed, the composite must have internal roles to which it can delegate messages that pass over these interfaces, unless, as discussed below, the composite provides such functionality itself rather than delegating it to a sub-role.

As we described in Chapter 5 (Figure 5.9), a functional role-player follows the principle of *blind communication.* The player is unaware of the structure of the organisation in which it plays a role. It is the external *role* that is aware of its local structure. While a role is aware of the external structure, a composite is aware of its own internal structure and acts as a message router. As shown in Figure 7-2 below (which extends Figure 5.9) a composite (Player1) receives incoming messages from the external role it plays (RoleA) then *delegates* these messages to the appropriate internal role (r1 or r2) depending on the type of message. In the ROAD framework, the routers in both roles and composites are implemented as Role-Message tables. Every role and composite has such a table. The records in these tables associate message types with role instances. The records in a composite's message-role table are updated by the composite's organiser as the composite is reconfigured. The implementation of these tables is discussed in more detail in Chapter 8.

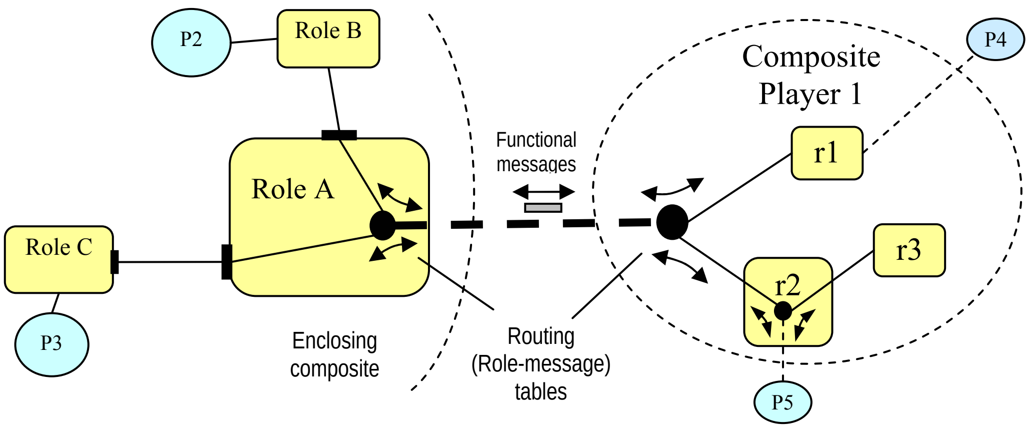


Figure 7-2: Self-managed composites delegates incoming messages to its roles

A number of issues arise with this conception of self-managed composites. The first is whether players that play the roles within a composite are also conceived as being "within" the composite, or external to it. Functional players are always separable from their roles, and may be separable from the composite. What 'separation' means here can vary. The players may be closely coupled to the implementation of the composite, for example, they may share the same deployment package. Or some players may have the power to create other players to play roles in the composite. The player may be in a different package, but be in a runtime environment that is shared with the composite. Or the player may be in a completely separate runtime and organisational context, as is the case if the player is a Web Service from an external provider. Given this range of possible 'degrees of separation', the ROAD framework is agnostic as to whether or not players are a *part* of the composite.

A second issue is whether the composite-player performs any domain function, or whether the composite delegates all function to the roles it aggregates. Related to this issue is whether or not the composite ("organisation") maintains state, as discussed in Chapter 5 (Section 5.6.2). In the current implementation of ROAD, composite players are purely structured containers that perform no domain function in themselves, and maintain no domain state[2]. This approach has an aesthetic appeal in that role and composite are the inverse of one another, as can be seen from Figure 7-2 above. However, this does not preclude an application developer extending the abstract composite class to include the implementation of domain functionality.

7.2. Organisers

Each self-managed composite has an organiser. Consistent with the principle of separation between process and control in the ROAD framework, the organiser function can be separated into an organiser role that provides the mechanisms (operations) to manipulate a composite, and a player that *controls* the reconfiguration.

Figure 7-3 below reproduces the illustration of our Widget Making Department composite from Chapter 4. The WidgetDepartment wd plays the WidgetMaker role instance wm in the Manufacturing Division composite. The WidgetDepartment composite has an organiser role wdo played by op1.

[2] In contrast to ROAD, PowerJava (Baldoni, Boella and van der Torre, 2005) extends the object-oriented paradigm and Java programming language with a pre-compiler to implement organisational roles and "institutions". This approach has many similarities to ROAD in that institutions (similar to ROAD composites) define roles that are played by players. A major difference in the approaches is that in PowerJava institutions perform domain functions and maintain domain state themselves. Institutions give 'powers' to the object playing the roles, and there is no organiser role.

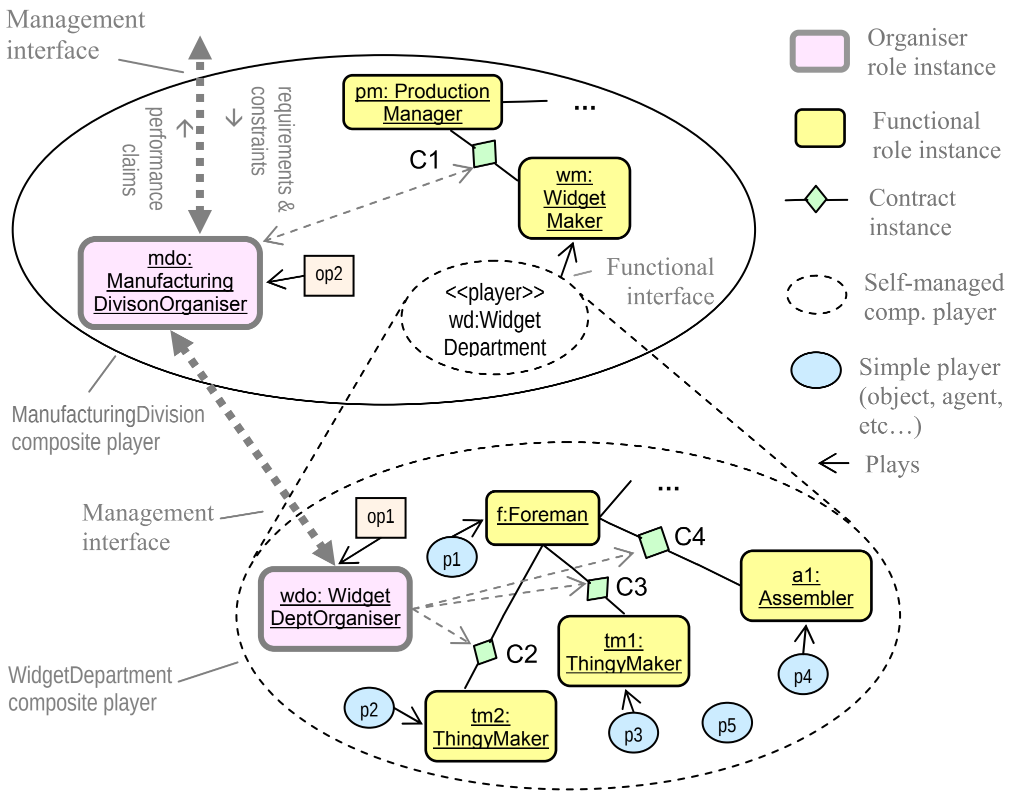


Figure 7-3: Example of recursive structure of self-managed composites

7.2.1. Organiser roles

The organiser *role* maintains a *representation* of its self-managed composite and has *operations* to manipulate and reconfigure the structure of its self-managed composite. These *internal* operations are standard to all organisers. Following the distinction made in Chapter 3, we can classify them as *reconfiguration* or *regulation* operations. These operations of an organiser *role* include:

Reconfiguration operations

- Create new instances of roles from predefined role types, and remove unwanted roles. For example, in Figure 7-3, if wd receives a request for more thingies, its organiser wdo can create an new ThingyMaker role tm2.
- Make and break contracts between roles in its self-managed composite. An organiser can instantiate, abrogate and reassign contracts between functional roles, e.g. wdo creates the contract c2 between tm2 and the Foreman f.
- Make and break bindings between its roles and players that are available and functionally compatible. e.g wdo binds player p2 to tm2, or changes tm1's player from p5 to p3.
- Maintain the composite's Role-Message (router) table so that incoming functional messages are delegated to the appropriate sub-role. In our

WidgetDepartment, the organiser wdo sets the Role-Message table so that all widget orders from the ProductionManager pm in the ManufacturingDivision are delegated to the Foreman f in the WidgetDepartment composite.

Regulation operations

- Monitor the *actual* performance of contracts in their self-managed composite. This can occur through the contract notifying its organiser of underperformance or breach. Alternatively, the organiser can poll its contracts.
- Change the state of a contract, e.g. from active to suspended, as described in Chapter 6.
- Update the *required* performance in the terms of a contract.
- Change the *conditions* in both general clauses and terms of a contract.

In order to carry out these internal operations an organiser must maintain a representation of the organisation. It must know:

- What roles and players/sub-composites it is controlling.
- What contract-types it has available to associate those roles.

It also needs to communicate with other organisers of any adjacent self-managed composites: i.e. the organiser of the enclosing composite, and the organisers of any sub-composites.

- Receive non-functional requirements from organisers higher up the management system e.g. wdo receives a request for an increased rate of production from the Manufacturing Division organiser mdo
- Transmit non-functional requirements to the organisers of any composite players under its control

7.2.2. Organiser players

The above operations and knowledge structures define a generic organiser *role* class. The role defines *how* to manipulate the composite. The responsibility of the player who plays the organiser role is to figure out *what* is to be done. This requires some level of decision making. Separating organiser roles from players allows generic functions to be separated from the domain-specific decision-making process. The decision-making functions that use domain-specific knowledge are implemented in the *player*. While, conceptually, the organiser player is separate from the organiser role, whether or not to implement them as separate runtime entities is a design decision. One advantage of decoupling the organiser role from its player is that players may vary in the intelligence they can apply to the decision-making process. For example, due to changing requirements an organiser player may be unable to reconfigure its composite in a way that fulfils its external obligations. In this case, the organiser can be upgraded to a more intelligent model such as a deliberative agent or human operator. These functions require some deliberative capacity. They include:

- Ability to select players. This requires the comparison of candidate player NFR characteristics (claimed and actual performance, availability etc.) and then deciding on the appropriate functional role-player bindings
- Ability to decide on appropriate configurations of its composite (i.e. what roles and contracts to instantiate) in order to remediate changing NFRs or changing performance of its players. Any such reorganisation must maintain the composite's viability. For example, if the structure is based on a bureaucracy, the organiser must ensure that proper chains-of-responsibility (i.e. supervisor-subordinate chains) are maintained to preserve the functional flow-of-control.
- Translation of non-functional requirements (NFRs) that are provided by the enclosing composite, to NFRs for contracts within the organiser's composite. These NFRs set the expected performance for a role under some term of the contract. For example, a WidgetDeptOrganiser player needs to translate a requirement for widget throughput into NFRs of contract terms related to rate of thingy production, as illustrated in Figure 7-5 below.
- As ROAD contracts are binary, a single contract cannot represent interaction dependencies between more than two parties (e.g. Party A must ask B to do x before it can ask C to do y). The organiser is responsible for coordinating such interaction. It does this by setting conditions in the form of executable assertions in the contracts. These assertions are dynamically updated to ensure any ordering constraints involving more than two parties are enforced.

The organiser player may also need to perform a number of functions that require interaction with the *external* environment. In the current state of the ROAD framework, these functions have not yet been implemented, but they are included here to complete the conceptual picture. They include:

- Ability to discover functional-role players that are candidates to play roles.
- Negotiation of service level agreements (SLAs) with external service providers. As pointed out in Chapter 6, internal ROAD contracts may be mapped to external SLAs when the player is, for example, a Web service provided by another business organisation.
- If the system is a control system that is dynamically responding to perturbations in the environment, unstable feedback loops can be created. The organiser may need to regulate NFRs to dampen such oscillations.

- The costs of reconfiguration may also need to be modelled by the organiser, so that any benefit gained from the restructuring is not lost due to, for example, time delays incurred by the restructuring process itself.

It should be noted that the above organiser-player capabilities only address non-functional reconfiguration and regulation as defined in Chapter 3 (i.e. ontogenic adaptation). Automated functional composition is a difficult problem and has been the focus of much research. While some of the dynamic architectures we discussed in the literature review in Chapter 3 propose ways to ensure functional compatibility at the syntactic level, the reliable composition of components that are semantically and interactionally compatible is still an open problem. Altough it is conceivable that a very smart artificial organiser could functionally construct composites, given the current state of research, we assume functional configuration is performed by a developer at design time. This means that the basic role and contract types that make up a composite type are predefined and cannot be changed at runtime. The framework allows for the reconfiguration of composites and the swapping of role-players in response to changes that lead to differences between required and actual performance. However, organisers in the current framework do not have the ability to compose *new* functions or create new *types* of association. A valid functional composition is assumed as a starting point.

7.2.3. Adaptive behaviour *within* a composite

An example of a generic decision-making process performed by an organiser is illustrated in Figure 7-4 below. Starting from the top-left of the flowchart, the organiser receives NFRs from the enclosing composite. It needs to translate these composite non-functional requirements (NFRs) into NFRs for terms in the contracts it controls. The organiser also monitors the actual performance of its contracts either actively (polling) or passively (waiting for a notification from the contract). If there is a mismatch between the actual and required performance (either because the requirements in the contract have been changed, or because the actual performance has decreased), the organiser attempts to mitigate this underperformance by reorganising its composite. The organiser chooses a strategy determined by the actual and claimed performance of the existing player and other available players. The *actual*-performance of a role-player pair is the historical performance measured as interactions pass through the contract. If a new player is allocated the role, then this actual performance data must be reset. In the absence of historical performance data, *claimed*-performance information could be obtained from a specification provided by the developer of the player or a third party accreditor.

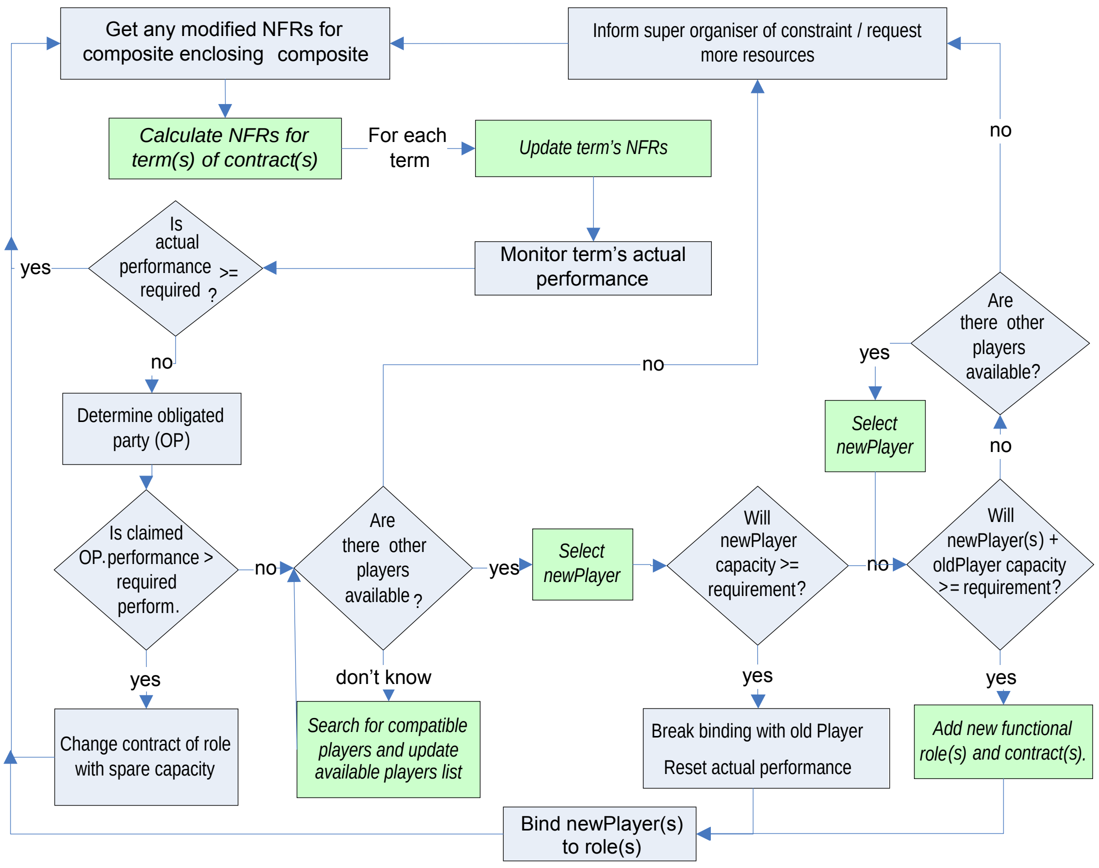

Get any modified NFRs for composite enclosing composite
Inform super organiser of constraint / request more resources
Calculate NFRs for term(s) of contract(s)
For each term
Update term's NFRs
Is actual performance >= required ?
Monitor term's actual performance
yes
no
Determine obligated party (OP)
Is claimed OP.performance > required perform.
no
Are there other players available ?
yes
Select newPlayer
Will newPlayer capacity >= requirement?
no
Will newPlayer(s) + oldPlayer capacity >= requirement?
no
Are there other players available?
yes
Select newPlayer
no
yes
Change contract of role with spare capacity
don't know
Search for compatible players and update available players list
yes
Break binding with old Player
Reset actual performance
yes
Add new functional role(s) and contract(s).
Bind newPlayer(s) to role(s)

Figure 7-4: Example of a decision making process of an organiser

In Figure 7-4 above, the types of function that *may* need to be implemented in an organiser *player* at the application-domain level (rather than the organiser *role*) have been italicised. The extent to which a player needs to implement the above functions will depend on the type of system being implemented. In general, the more *open* the system (that is the more complex the cybernetic *variety* (Ashby, 1956) of the composite's environment), the more *capability* the organiser-player will need to maintain the viability of its composite.

7.3. The management system

In the previous sections of this chapter, we have focused on the adaptive behaviour of a *single* composite. However, as pointed out above, the organiser of a composite receives non-functional requirements from the organiser of the enclosing composite, and transmits NFRs to the organisers of any composites that play roles under its control. This hierarchical network of organisers (the thick dotted lines in Figure 7-3 above) constitutes a *management system*[3] that can be described and controlled independently from the players that interact directly with the application domain.

This network of organisers is analogous to the management-systems that exist in man-made organisations. For example, the management structure or financial system in a manufacturing business can also be described at a separate level of abstraction from the functional processes that transform labour and material into products. Management-systems maintain some form of representation of the requirements and current state of the underlying functional system. These models will vary depending on the variables that need to be controlled in order to maintain the system's viability (ability to survive and fulfil its function) in its environment, as we have previously discussed in Section 2.3.

An organiser provides the management interface to its composite and interprets the regulatory control messages that flow through this network into terms for the contracts within its composite. These messages are non-functional requirements expressed in terms of metrics. Each metric is associated with a utility function, and these utility functions are in turn associated with contract terms, as shown in Figure 6-3 of Chapter 6. Non-functional performance requirements flow down the management hierarchy, and information on the performance of the managed composites (actual or claimed) flows up. As defined above, it is the organisers' responsibility to translate these messages into NFRs that are applicable to the contract terms in the various contracts it controls.

We will call these two regulatory control-message flows, respectively, top-down *requirement/constraint-propagation* and bottom-up *performance-monitoring*. We will illustrate their dynamics in the next section.

7.3.1. Adaptive behaviour *across* composites

The structure and adaptive behaviour of a management-system will be illustrated with our WidgetDepartment example. As shown in Figure 7-3 above, the WidgetDepartment (wd) plays the WidgetMaker role within the ManufacturingDivision. The relationship between functional requirements and NFRs is illustrated with a production scheduling problem. We need to keep in mind that in an open system, the time taken to execute a function may vary or come at a cost.

[3] In (Colman and Han, 2006) we refer to this management system as a *coordination* system. We have changed the terminology here to emphasise that this system involves more than just synchronisation, for example it handles QoS concerns.

Requirement and constraint propagation.

Performance requirements pass down the hierarchy of organiser roles to alter the performance requirements of the contracts. In our ManufacturingDivision the ProductionManager receives (from above) orders for widgets. It determines the priority of the orders, and passes these on to the WidgetMaker role (as determined by the contract C1). The ManufacturingDivisionOrganiser (mdo) receives performance requirements and constraints, and in turn adds derived NFRs to the contract C1. For example, the contract may require that widgets be made within certain time constraints, or that certain resource costs not be exceeded. The WidgetMaker role is played by the WidgetDepartment (wd) self-managed composite. The performative level of contract C1 allows the ProductionManager pm to invoke the WidgetDepartment do_widgetOrder(...) method. To fulfil its obligations under the C1 contract, the wd must organise the production of Widget components, i.e. thingies and other parts (not shown) at a rate specified in the contract.

The Foreman f is a *delegate* (as discussed above in Section 7.1) for the WidgetDepartment's interactions with the WidgetMaker role, and provides the implementation of the composite's functional interface. Via this interface, the Foreman f receives orders for widgets, and in turn allocates work to, among others, the ThingyMakers (tm1, tm2 etc as shown in Figure 7-3). The Foreman can do this under the terms of the Foreman-ThingyMaker contracts (instances C2 and C3) by invoking ThingyMaker's `do_makeThingy()` method. While the contracts C2 and C3 have the same form, these instances of the Foreman-ThingyMaker contract have different performance characteristics written into their respective contract schedules. Suppose the role-player attached to tm1 has a *claimed* performance of 10 thingies per hour, while the role-player attached to tm2 claims only to make 5 thingies per hour. The Foreman f (party to contracts C2 and C3) can use this performance capacity information in the schedule when deciding to whom the work should be allocated.

Required-performance information on goals and constraints is transmitted through the organiser roles. In Figure 7-5 below, the organiser role of the WidgetDepartment composite (wdo) receives NFRs from the organiser above it in the management hierarchy (mdo). These NFRs (which are stored in contract C1) include the requirement that the player playing the WidgetMaker role has a certain throughput of widgets. The organiser of the sub-composite (wdo), then reinterprets these into measurable performance requirements for the contracts it controls (e.g. C3). To achieve the required widget throughput, it follows that there must be sufficient rate of thingy production.

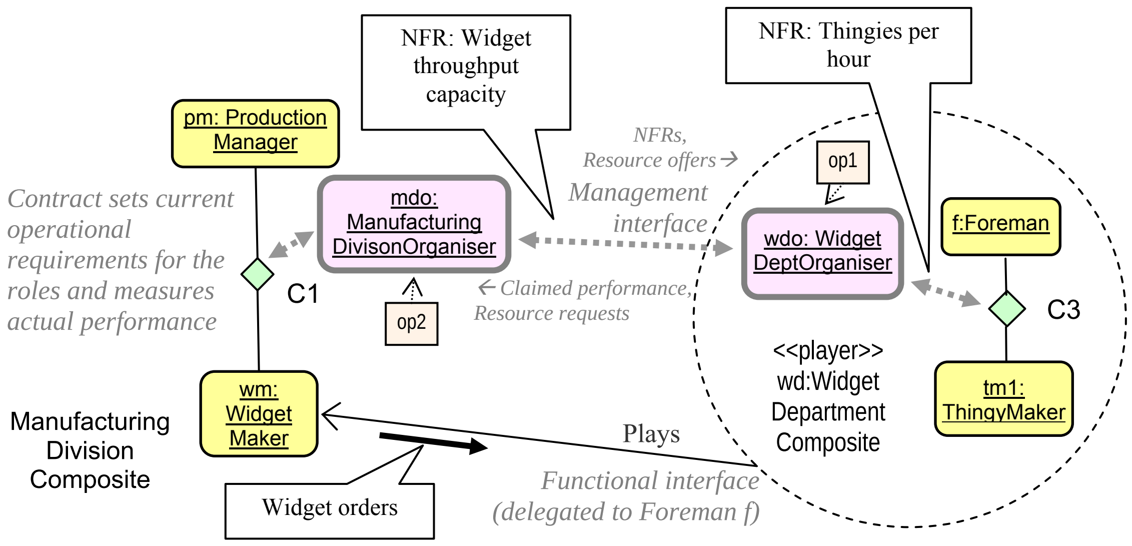

NFR: Widget throughput capacity
NFR: Thingies per hour
pm: Production Manager
mdo: Manufacturing DivisonOrganiser
NFRs, Resource offers →
Management interface
op1
wdo: Widget DeptOrganiser
f:Foreman
Contract sets current operational requirements for the roles and measures actual performance
C1
op2
← Claimed performance, Resource requests
C3
<<player>> wd:Widget Department Composite
tm1: ThingyMaker
Manufacturing Division Composite
wm: Widget Maker
Plays
Widget orders
Functional interface (delegated to Foreman f)

Figure 7-5: Interactions between an organiser role (wdo) and the organiser of enclosing composite (mdo).
(Detail from Figure 7-3 on page 123)

Note that these NFRs flow over a *management network* of organisers and contracts (dotted arrows Figure 7-3 and Figure 7-5). This network is distinct from the *functional network* over which the process is enacted. The functional structure is a network of functional roles and their players (i.e. solid lines in Figure 7-5 representing the functional role-player binding and functional role-role relationships). These two networks intersect at the contracts, which are points at which management exerts it regulatory control.

Performance monitoring and breach escalation.

The organiser role can monitor the contracts to see if they are meeting their performance requirements. Changing requirements, environment or computational contexts can lead to the violation of performance clauses in the contract. If the *actual* performance (as recorded in the contracts) falls below the *required* performance, then the organiser must attempt to reconfigure the composite by altering the existing contracts, reassigning roles to more capable players or creating new contracts and roles. For example, if the combined output of tm1 and tm2 (in Figure 7-3) cannot meet the required performance of the composite (as determined by contract C1) then the WidgetDeptOrganiser wdo needs to reconfigure its composite using a strategy similar to that outlined in the flowchart Figure 7-4 above.

If this reconfiguration is beyond the current capability of the composite organiser, then the problem is escalated up to the next level. The organiser at the higher level must then try to mitigate the underperformance. In our example, it could replace the

WidgetDepartment wd with another player that can play the WidgetMaker role (e.g. an outsourced service), or alternatively create another WidgetMaker role and player to take some of the load.

The above organiser operations are necessary for a composite to meet changing *operational* requirements. However, as pointed out in Chapter 2, adaptive management may also include higher levels of control that involve anticipation of the future and the planning of changes (Systems FOUR and FIVE in the Viable System Model (Beer, 1979)). For example, in Figure 7-3 above, if wdo detects that the thingyMakers will be over-loaded, it may ask mdo for resources to get more thingyMakers. Such escalation can be viewed as an organised form of exception-handling, where performance messages flow up through the *management*-system before failure occurs in the functional system. Just as an animal detecting a threat will increase its adrenaline levels to stimulate the heart rate for flight or fight, so the management-system detects stress on the system then changes the parameters of contracts (or reorganises them) in an attempt to avert system failure. Such proactive management would involve *capacity planning*, where management tries to estimate the future demand and then tailor the system's configuration to meet that demand. This would require that higher-level organisers have the ability to query lower-level organisers as to the capacity of their composites given particular resource constraints. These lower-level organisers would need to calculate such capacity, a process that may involve querying the next level below, and so on. To achieve this anticipatory adaptivity, organiser player would need the ability to reason about capacity and resources particular to a domain. Protocols between organiser roles that would permit negotiation of NFRs and constraints would also be required. The examination of such advanced adaptive techniques provides a promising area for future research, but is beyond the scope of this thesis.

7.4. Summary

In this chapter we have described how role, players and contracts are structured into self-managed composites. These composites are themselves role-players which enables an application to be structured as a reconfigurable recursive hierarchy. Composites hide their internal structure by delegating incoming messages to appropriate internal functional roles. Each composite has an organiser role. Organiser roles provide the means by which composites can be restructured by creating new functional roles, by creating contract between roles, or by changing the binding between functional roles and their players. Organiser roles also regulate the composite by monitoring and changing the clauses in the contracts they control. Organiser roles have players who are responsible for deciding what reconfiguration or regulation is necessary as requirements or conditions change. Organiser players are separable from their roles, and may be of varying capability.

Organiser roles provide a management interface to their composite. The organisers of composites in a recursive hierarchy are connected via these interfaces. This network of organisers and the contracts they control constitutes a management system over which non-functional requirements and performance information pass. This network facilitates the operationalisation of goals down through the structure, and the escalation of any performance deficits or problems that lower level composites can’t handle to higher levels.

Treating the management system of a software application as a concern separate from the functional system, allows to us to represent the application at a higher level of abstraction. This organisational level of abstraction allows us to modulate the behaviour of the structure by dynamically representing and manipulating non-functional attributes and performance over that structure. By representing software at this organisational level, we may also be able to start to build a “management theory” of software that can help us design and evaluate software.

To conclude this chapter and this part of the thesis, the UML diagram of ROAD concepts in Figure 7-6 below provides an overview of the ROAD concepts discussed in Chapters 4 to 7. These concepts are the meta-model on which the ROAD framework is based. The implementation of this meta-model as a class framework is described in Part III of the thesis.

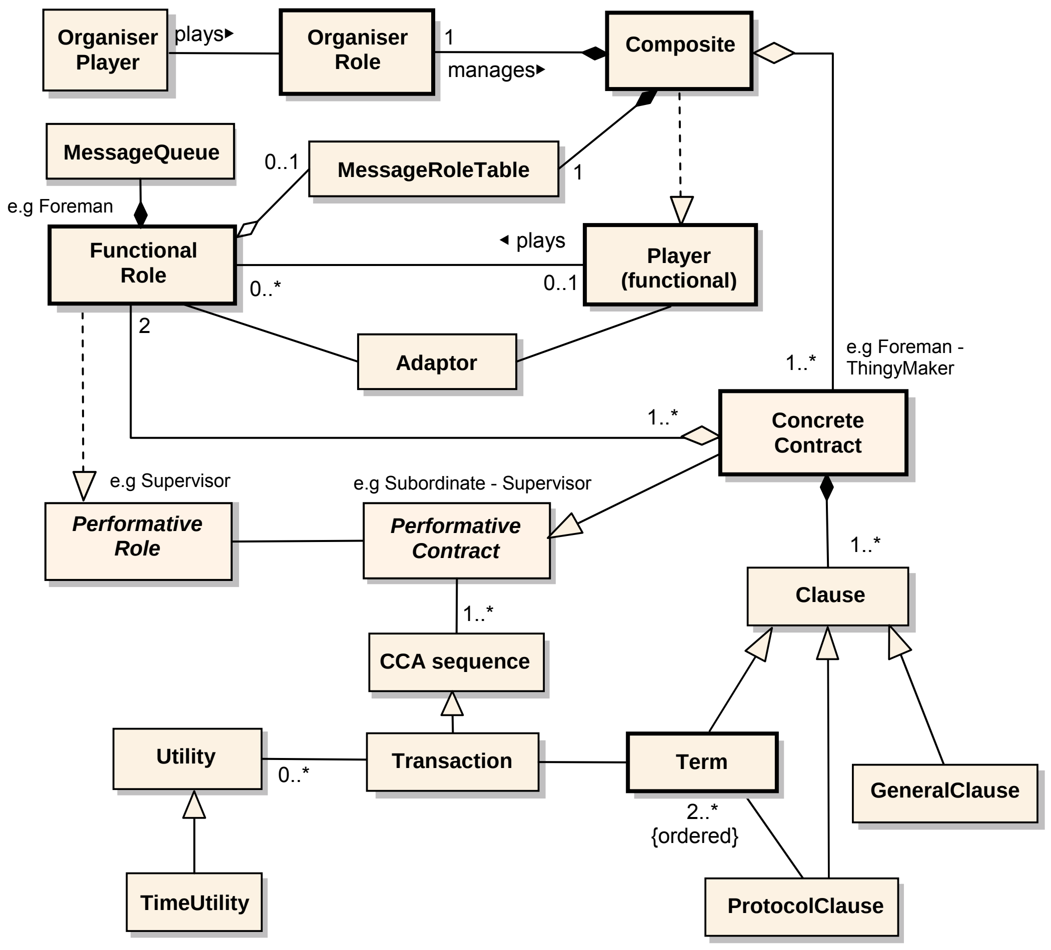


Figure 7-6: Summary of relationships between ROAD concepts

Part III

The ROAD Framework and Discussion

8

Framework Implementation

The ROAD conceptual model, described in the previous part of this thesis, defines a flexible structure that supports indirection of association and instantiation, and provides a way to manage such indirection. The approach we have taken to creating adaptive systems is to develop a domain-independent framework that implements the ROAD meta-model. The application developer extends this framework by writing domain-specific organisational code. In this chapter we describe the framework and some key aspects of its current proof-of-concept implementation. An application built on this framework is then described in the following chapter (Chapter 9).

The implementation of the ROAD framework not only has to provide a scaffold on which the developer can structure the specific application, but needs to do this in a way that is practical for the developer, and does not introduce unnecessary dependencies that reduce adaptivity. The ROAD framework reduces the load on the application developer by providing reusable abstract classes that hide the complexity of the adaptive mechanisms in ROAD. Unnecessary dependencies are avoided by maintaining a separation of concerns between organisational code and functional code. These two concerns can be independently modified, and the organisational constructs can be superimposed *post facto* onto the functional code.

In this chapter we begin by describing the framework-based approach we use to implement adaptive software, and then we give an overview of how the concepts described in the previous part of the thesis map to key abstract classes in the framework. The subsequent sections then describe in more detail the Java implementation of these key ROAD concepts; namely, roles, contracts, self-managed composites and organisers. In particular, we describe a novel use of a type of instantiable aspect called an “association-aspect” to implement contracts at the performative and functional levels. We then discuss the work that could be done to further develop the ROAD framework and what tool support is needed by developers to make practical the development of adaptive software organisations.

8.1. A framework-based implementation

Our implementation of the ROAD framework separates code into independent packages: domain-independent organisational code; domain-specific organisational code; and domain-specific functional code. Figure 8-1 below illustrates these three concerns:

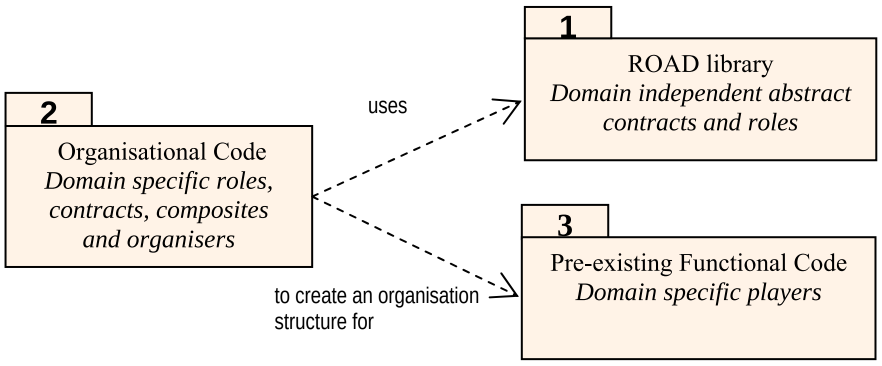

1
ROAD library
Domain independent abstract contracts and roles
2
Organisational Code
Domain specific roles, contracts, composites and organisers
uses
3
Pre-existing Functional Code
Domain specific players
to create an organisation structure for

Figure 8-1: Organisational code based on the ROAD library package can be written as a separate concern from the functional code

1. **Domain-independent organisational code** is a reusable library that defines abstract classes for functional roles, contracts, organiser roles, composites and utility functions. This library also defines abstract performative contracts that enforce prototypical interaction control patterns (e.g. a Supervisor-Subordinate contract). If the abstract performative contracts provided by the ROAD library do not express the necessary control relationships, new abstract contracts can readily be defined.

2. **Domain-specific organisational code.** The programmer creates domain-specific roles, contracts, composites and organisers based on the abstract classes in the ROAD library package. Domain-specific utility function classes are also written to extend the abstract utility class. These utility functions allow performance requirements to be specified and enforced for interactions between the particular functional roles that are bound by the contract. Domain-specific organiser players are also created to implement specific adaptation strategies.

3. **Domain-specific functional code** consists of entities that will play the roles in the organisation. Classes representing these can be written without defining the configuration of the organisation in which they will participate. As players have to be

compatible with the roles they play, it may also be necessary to write adaptors that can resolve any mismatch between role and player interfaces. As ROAD provides a heterogeneous framework, it may also be necessary for these adaptors to translate between implementation technologies, for example between the Java invocations in the ROAD organisation, and the SOAP messages of a Web-service player.

The organisational code from the organisational packages is compiled using a modified AspectJ 'association aspect' compiler (Sakurai, Masuhara, Ubayashi et al., 2004)[1] to create an adaptive application. The roles in this application are then bound at runtime to their players. Figure 8-2 below is an implementation version of the conceptual model presented in Figure 7-6 at the end of the previous chapter.

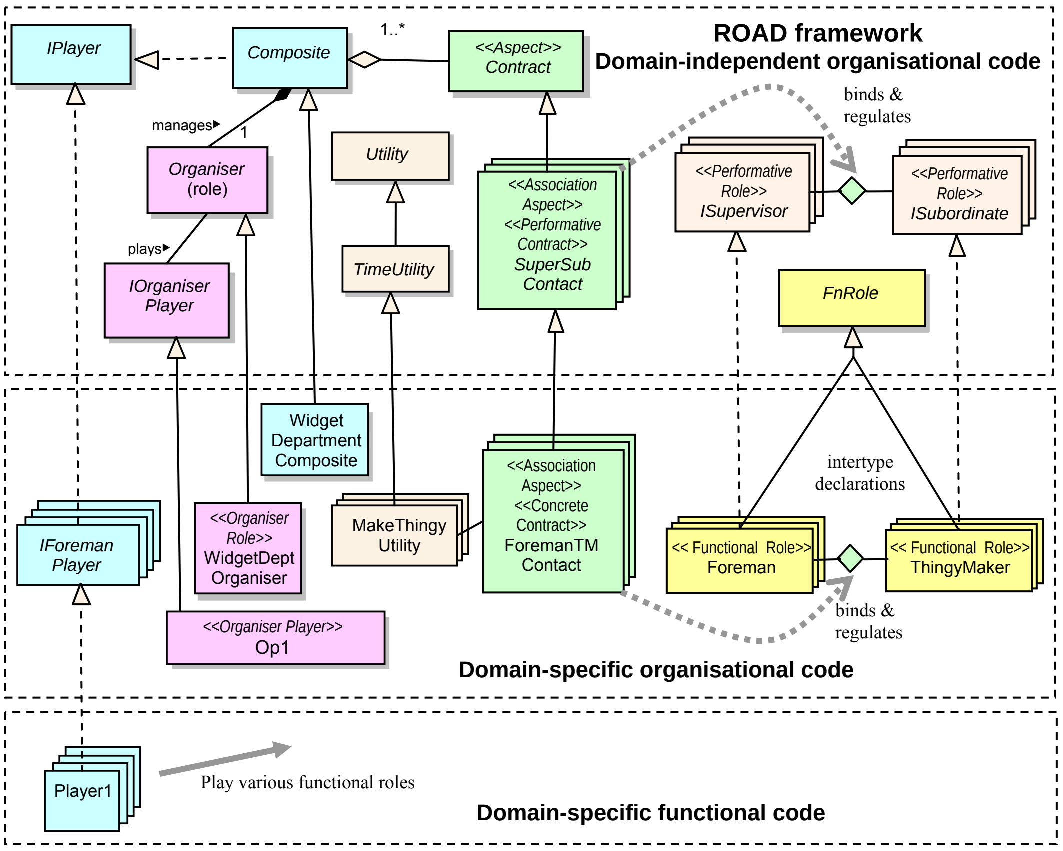

ROAD framework
Domain-independent organisational code
IPlayer
Composite
1..*
<<Aspect>> Contract
binds & regulates
manages
1
Organiser (role)
Utility
<<Association Aspect>> <<Performative Contract>> SuperSub Contact
<<Performative Role>> ISupervisor
<<Performative Role>> ISubordinate
plays
IOrganiser Player
TimeUtility
FnRole
Widget Department Composite
intertype declarations
IForeman Player
<<Organiser Role>> WidgetDept Organiser
MakeThingy Utility
<<Association Aspect>> <<Concrete Contract>> ForemanTM Contact
<< Functional Role>> Foreman
<< Functional Role>> ThingyMaker
binds & regulates
<<Organiser Player>> Op1
Domain-specific organisational code
Player1
Play various functional roles
Domain-specific functional code

Figure 8-2: The ROAD Framework library and an example domain-specific application

The class diagram of the *domain-independent* organisational ROAD library only shows the abstract classes and interfaces that application extends with *domain-specific* code. The other supporting classes that were shown in the conceptual diagram (e.g.

[1] The Association Aspect compiler and source code is available for download at http://www.komiya.ise.shibaura-it.ac.jp/~sakurai/aa. The provided installer installs the compiler and runtime library in the same manner as AspectJ. Programs are compiled in an identical manner to AspectJ by using the ajc command. At the time of writing the Association Aspect compiler was based on AspectJ v1.2.

MessageRoleTable) will be discussed in the subsequent sections. The figure also shows how a domain-specific organisation, in this case the WidgetDepartment composite from our running example, is created by extending the ROAD framework. Java interfaces are denoted by the class name prefixed with an "I" e.g. `IPlayer.` The domain-specific functional code (i.e. the players) can be created and compiled independently from the organisational code.The following sections will look at the above model in more detail.

8.2. Roles

In Chapter 5 we defined the concept of an organisational *functional* role. To recap, the properties of such roles are:

- A functional role is a first-class entity that defines a position within an organisational structure. It defines an *abstract function* that is the aggregation of its obligations set out in the terms of the contracts to which it is party.
- Functional roles can also be classified as playing a performative role with respect to each particular role-role relationship in which they participate (as shown in Figure 8-2 above a functional role inherits from an empty performative role interface that allows them to be bound into performative contracts).
- A role provides a *router* so that outgoing messages from its player can be sent to the appropriate associated role (contracted party).
- Roles provide a *message queue* for incoming messages so that the role's function is still viable in the case of its player being temporarily unavailable.
- Roles may also need to have *adaptors* added to enable them to talk to players of various types.

The ROAD framework provides an abstract class `FnRole` that provides functionality such as keeping track of the contracts to which the role is a party, message queuing and routing. We will now look at how each of these properties has been implemented in the ROAD framework.

8.2.1. Roles as abstract function.

As described in Chapter 5, functional roles present two categories of interface. These interfaces are illustrated in Figure 8-3 below. The first category of role interface is associated with another role via a contract (`f1` and `f2` in the diagram). The totality of these contracted interfaces represents an abstract function of the role *with respect to* its organisation. The other type of role interface is its interface to its player (`f3` in the diagram). Remember that in ROAD the role performs no domain function in itself, but rather all process is executed by the player. The role-player interface `f3` is, therefore, merely a re-expression in complementary form[2] of the role-role interfaces (`f3` = Σ(`f'1,f'2`) where Σ is some aggregate re-expression and `f'n` is the complement of the interface `fn`). The interface the *player* presents to its role (`f4`) can therefore be regarded as some form of aggregation of its role's organisational interfaces (Σ(`f1,f2`)).

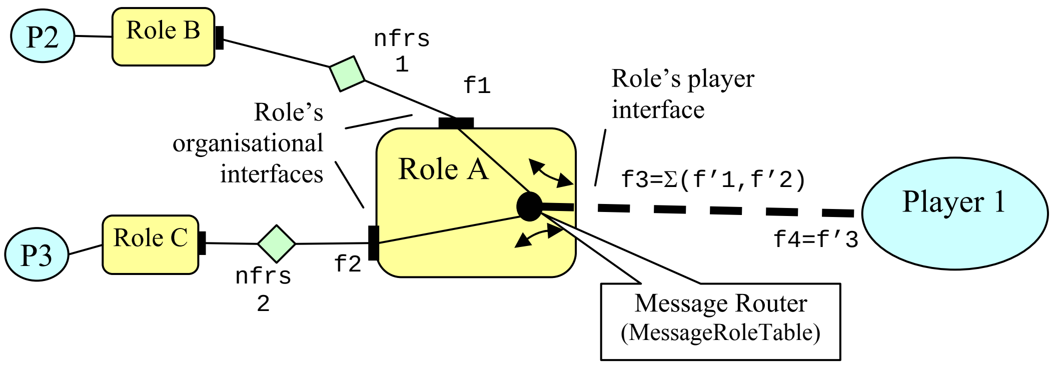


Figure 8-3: Relationship between role and player interfaces

If these interfaces only express the *syntax* of required and provided method signatures, then the aggregation Σ is straightforward: the role merely retransmits the message. However, as described in Section 5.4, a functional role may also specify performance requirements, interaction protocols, authority relationships and access to resources. In ROAD these non-functional requirements (NFRs) are defined in contracts that bind the role (nfrs1 and nfrs2 in Figure 8-3). The implementation of contracts is discussed in the next section. However, it is possible that the desirable NFRs for a role may be conflicting and need to be balanced against each other. This aggregation of NFRs is a task of the composite's organiser (as described in Chapter 7). The current implementation of roles does not express an aggregated "position description" of non-functional requirements that could be used to automatically match compatible roles and players. We leave the development of NFR aggregation, and automatic search and selection to future work. However, the inability to match NFRs is not necessarily fatal in ROAD. An advantage of the ROAD approach is that its contracts enact exogenous measurement of performance, and can also enforce protocols. A player can always be bound to a role on a "suck it and see" basis, even if its performance claims or characteristics are uncertain. If the actual performance of the player is satisfactory, in terms of the role's contracts, it will be retained. If not, alternative players could be sought.

[2] Where required interfaces become provided interfaces, and vice versa

8.2.2. Roles as message routers

In ROAD, it is the roles rather than the “blind” players that represent the structure of the organisation. A role needs to be aware of the contracts that bind it, and needs to be able to route outgoing messages from its player to the appropriate associated role (as shown in Figure 8-3). We implement this structure by means of a MessageRoleTable which contains a dynamic number of MessageRoleRecords as shown in Figure 8-4 below. Each record associates a message signature with one or more role instances that can handle that message. For example, a Foreman role may have an entry in the table that associates a `do_makeThingy()` method invocation with objects of type ThingyMaker. In some circumstances a role might be contracted to multiple instances of roles of the same type (e.g. in our example shown in Figure 7-3 the Foreman role instance `f` has contracts with two ThingyMaker roles, `tm1` and `tm2`). Such multiplicity of role relationship is allowed if the Boolean field isMultipleAllowed is set to true. Where there are a number of possible role instances to which a message can be sent, some sort of allocation scheme is required to determine which role instance receives the message. Allocators are created by the application developer to implement a method that returns the appropriate role to which the request will then be allocated. Allocation schemes can be simple (a default round-robin scheme is provided by the framework) or can be more complex domain-specific schemes based on the capabilities of the players. More details on our implementation of message allocation can be found in (Pham, Colman and Han, 2005).

.

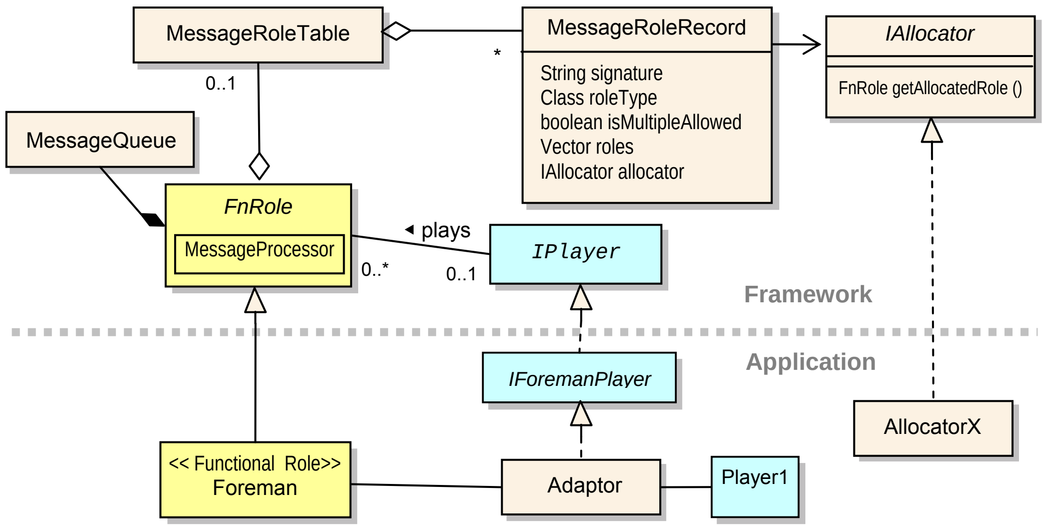


Figure 8-4: Role related entities

Each entry in a MessageRoleTable reflects a term in one of the contracts to which the role is a party. Entries are maintained by the composite's organiser as it adds and removes contract terms.

8.2.3. Message queuing in roles

As discussed in Chapter 5 (5.6.1), roles are also responsible for preserving communication state, thus ensuring that the organisational structure remains viable when a player is temporarily unavailable, e.g. during the swapping of a role's player. Messages coming into the role are placed in an incoming queue. Each role has an inner MessageProcessor class as shown in Figure 8-4. When the player becomes available, the MessageProcessor extracts the message from the FIFO queue and sends it to the player. Because the current implementation of ROAD is written in Java, method invocations to the role have been encapsulated into messages so they can be stored. This approach has been adopted, rather than using message-oriented middleware, because it accords with the aspect-based method interception used in the ROAD contracts.

The use of incoming queues accords with a "push" or *command-driven* form of organisation; that is, an organisation where process is executed by the flow of commands down through the organisational structure. Using our example, the Foreman role would invoke a `do_makeThingy()` operation in the ThingyMaker. The Foreman does not have to worry whether or not the ThingyMaker role has a player attached, because the message will be stored in the ThingyMaker's queue (its "in-tray"). Such incoming queues can be bounded so that the ThingyMaker does not receive too many "pending" tasks. It follows that, in a push model, there needs to be some sort of notification back to the requester if the in-queue is full[3].

An alternative to the push mode of organisation is a "pull" model. A pull-model is *demand-driven*. Request messages are placed in a pool (an out-queue on the role), and retrieved by the role(s) that service that request when they are ready. The demand-driven mode of organisation removes the need for the requestor role to keep track of the service role's in-queue so that it does not become too full. It also obviates the need for a proactive allocation scheme if there is more than one role of the same type (e.g. multiple ThingyMakers). However, there are some complications associated with the

[3] In a push model the issue arises of what happens to the sent messages if the role fails. It might be thought that some form of compensation scheme is necessary to retrieve the sent messages, and restore communication state to what it was prior to the failure. However, we would argue that this is a non-issue as roles do not fail (or if they do then the whole composite has failed) – it is their players that fail. If the player fails, it is swapped with a new one and communication state is preserved (with perhaps the exception of the transaction that is currently being processed).

pull-model. These include the need for multiple out-queues (one is needed for each type of message type); difficulties in request and response matching; and the arbitrariness in allocation (first-in, first-served) of parameterised requests that may require different processing capacities. For these reasons, only the push-model is currently implemented in ROAD. An extended discussion of the relative merits of push and pull models in ROAD can be found in (Pham, Colman and Han, 2005).

8.2.4. Role-player adaptors

A major advantage of the ROAD approach is that it allows the possibility that heterogeneous players (objects, components, services, agents, etc) can play roles in a ROAD organisation. Rather than alter the role to match the technology of the player, adaptors are added to roles to match the particular technology of the player. At the syntactic level, ROAD role interfaces are Java calls. If the player is, for example, a Web service, then an adaptor is needed to convert between Java calls and SOAP messages. Similarly, different syntactic/technology level adaptors would be needed if CORBA components and agents are used. In its current state of implementation, the ROAD framework supports players that are Java objects and Web services. These adaptors are proxies for the players, and implement an interface that represents the player, as shown in Figure 8-4 above. From the role's point of view, an adaptor looks no different to a player.

As we have pointed out above, interfaces ideally should describe more than relationships at the syntactic level. Semantics, behaviour and QoS need also to be represented if complete compatibility of entities is to be assured (Han, 1998). Although behavioural and QoS levels are currently supported in internal ROAD role-to-role contracts, the mapping of these qualities to the role-player relationship is outside the scope of this thesis. Extending ROAD adaptors to handle behavioural protocol mismatches between the role and the player is a current area of research.

8.2.5. The role life-cycle

Instances of roles are created and destroyed by the organiser of the composite to which they belong. Currently the role types available to an organiser are statically defined Java classes, and role objects are dynamically created in the same way as any Java object[4]. As the role is made party to contracts (or removed from contracts) its list of

[4] However, from the discussion in the previous section, it will be apparent that everything we need to know about a role, other than its name, can be derived from the contracts that the role participates in. A role is a named position (in a role-structure) that aggregates obligations contained in the contracts which bind it. The actions of a role (message routing, queuing) is either generic or externally added (adaptors). Role-types should therefore be able to be dynamically created from textual descriptions of its contracts. This is a task for future work.

contracts and its MessageRoleTable are updated. Similarly, as a player is bound or removed from the role, the role's references to the player is updated (and vice-versa) by the organiser. A number of issues related to integrity of the role-structure arise. Role instances need to exist in order to create a contract between them, but what happens if the organiser wants to delete a role that is bound to one or more contracts? Should this be prevented or should there be an automatic cascade deletion of the contracts? Can a role instance exist with no current contracts? Such issues are not currently addressed in the ROAD framework, but might be viewed as constituting an organisational "style" which may vary depending on the problem domain.

8.3. Contracts

In ROAD, all organisation-related interactions between roles, and thus by extension their players, pass through a contract that associates those roles. As described in Chapter 6, instances of ROAD contracts perform a number of functions. A contract:

- Creates a connector between roles
- Defines contract terms that
    1. control the types of message that can flow over the connector in either direction
    2. define transactions as represented by messages passing between the roles. These transactions can be of various types (e.g. synchronous, asynchronous, etc.)
    3. define performance measurement points associated with different types of transactions
    4. allow arbitrary utility function objects to be attached to these measurement points
    5. sets required performance in terms of the metrics defined in those utility functions
    6. measure actual performance using only those utility function objects.
- Can define other types of clause that set conditions on its existence (general clause) or define required sequences of transactions between the parties (protocol clause).

In this section we will describe an implementation of the ROAD concept of contract that uses *association-aspects*. Instances of these aspects allow contracts to be defined that associate two[5] role instances, and to intercept the messages that pass between these

[5] Association aspects can associate more than two objects, but in ROAD we limit contracts to binary associations for the reasons discussion in Chapter 6.2.

roles. Association-aspects can also support contract abstraction as described in Chapter 6.

Figure 8-5 below presents a model of ROAD contracts showing some of the major contract related classes and their properties. This model is an implementation version of the conceptual model of contracts shown in Figure 6-3.

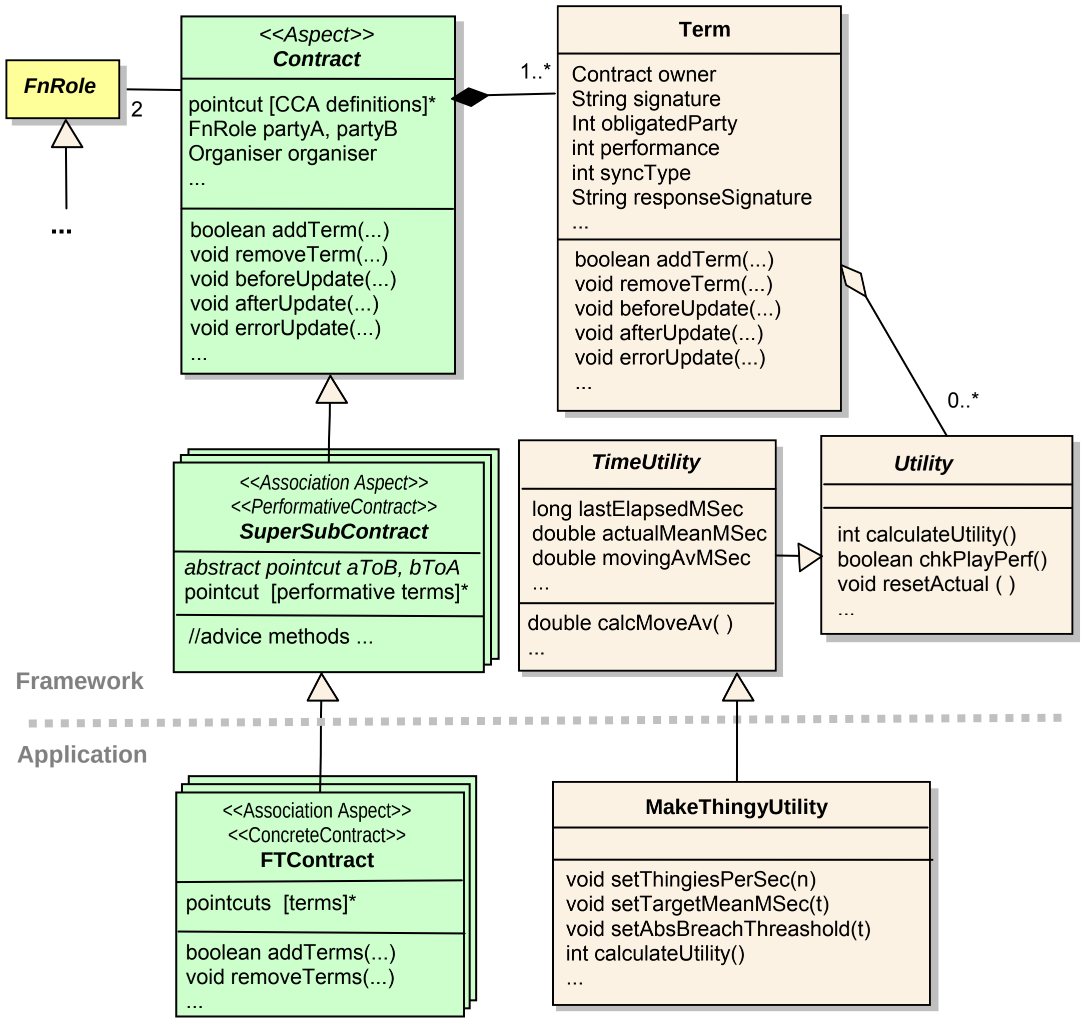


Figure 8-5: Contract related classes

This model will be elaborated in the sections that follow. The current state of implementation of the ROAD framework does not yet include general and protocol clauses as described at the conceptual level in Chapter 6. We begin by introducing association-aspects in the context of aspect-oriented technology and show how they can associate a group of objects into a contract. We then show how abstract performative contracts (abstract association-aspects) can be created using association aspects. These abstract contracts 'intercept' method calls based on their CCA (control-communication act) type. In ROAD, interaction is controlled and monitored at this abstract level. We then discuss in more detail the inheritance hierarchy of *contracts*, and the related classes of *terms* and *utilities*. We conclude this section on contracts with a discussion of the limitations of the current implementation of contracts using association-aspects.

8.3.1. Creating contract instances using association-aspects

Aspect-oriented methods and languages (Kiczales, Irwin, Lamping et al., 1997) seek to maintain the modularity of separate cross-cutting concerns in the design and source-code structures. Examples of cross-cutting concerns that have been modularised into aspects include security, logging, transaction management and the application of business rules. The AspectJ (Eclipse Foundation, 2004) extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow such as a *call* to a method). An *advice* is code that is executed when a join point that matches a pointcut is reached. *Aspects* encapsulate such pointcuts and advices. These units of modularity can model various cross-cutting concerns. A short glossary of the key AspectJ terms is provided in Figure 8-6.

An Aspect J Glossary

AspectJ

AspectJ extends the Java language with constructs that allow the encapsulation of concerns (aspects) that crosscut standard classes. AspectJ provides a compiler that *weaves* aspect code through the base code at compile-time. As of version 1.5 post-compile (binary) and load-time weaving are also supported.

Joinpoint

A *join point* is a well-defined point in the program flow. AspectJ can access a variety of join points including method call and execution, constructor call and execution, read or write access to a field, and exception handler execution.

Pointcut

A *pointcut* picks out certain join points and values at those points. For example, the pointcut

`call(void Point.setX(int))` picks out each join point that is a call to a method that has the signature `void Point.setX(int)` — that is, Point's `void setX` method with a single int parameter. Pattern matching can be used to pick out join points. Pointcuts can also be named and composed from other pointcuts using the && (and), `||` (or), and `!` (not) operators.

Advice

A piece of *advice* is code that is executed when a join point defined by a pointcut is reached. In AspectJ advice can be run just *`before`* or *`after`* a pointcut is reached at runtime. Because Java programs can leave a join point 'normally' or by throwing an exception, there are three kinds of after advice: `after returning`, `after throwing`, and `after` (which covers both the other cases). `Around` advice allows alternative code to be executed.

Aspect

An *aspect* brings together a *pointcut* and an *advice* to define aspect implementation. The aspect runs advice at join points picked out by the pointcut.

Figure 8-6: Glossary of AspectJ terms

While AspectJ-like aspects have previously been used to add role behaviour to a single object (Kendall, 1999), as far as we are aware they have not been used to implement associations between roles. Aspects, as they are currently implemented in AspectJ, do not easily represent the behavioural associations between objects (Sullivan, Gu and Cai, 2002). While current implementations of AspectJ provide *per-object* aspects, these have to associate a unique aspect instance to either the executing object (*perthis*) or the target object (*pertarget*). When an advice execution is triggered in an object, the system looks up the aspect instance associated with that object and executes that instance. The per-object approach allows the aspect to maintain a unique state for each object, but not for associations of groups of objects. In order to implement contracts, we need aspect instances that bind groups of objects, and that can be created and destroyed in the same way that objects are created and destroyed.

Sakurai et al. (Sakurai, Masuhara, Ubayashi et al., 2004) developed *association-aspects* to enable an aspect *instance* to be associated with a group of objects[6]. *Association-aspects* meet the expressive requirements for implementing contracts as defined above, and are implemented with a modification to the AspectJ compiler.

Association-aspects allow aspect *instances* to be created in the form

```
MyAssocAspect a1 = new MyAssocAspect (o1, o2, … , oN);
```

where a1 is an aspect instance and o1…oN are a tuple of two or more objects associated with that instance. Association-aspects are declared, as in Listing 8-1 below, with a *perobjects* modifier (rather than *perobject*) that takes as an argument a tuple of the associated objects. The ability to represent the associative state between objects in a group makes association-aspects suitable for representing contracts as we have defined them (Colman and Han 2006c). Using our running example above, the declaration of a concrete contract to bind two objects of type Foreman and ThingyMaker would be as follows:

```
public aspect FTContract extends SuperSub perobjects(Supervisor,
     Subordinate){
  private Foreman f;      //implements the Supervisor interface
  private ThingyMaker t;  //implements the Subordinate interface
  public FTContract(Foreman f, ThingyMaker t)  {
   associate(f, t); //creates the association-aspect
   this.f = f;
   this.t = t;
   ...
  }
  //contract term pointcut definitions and advice
   ...}
```

Listing 8-1: Declaring a concrete contract

[6] In the AOP language Eos (Rajan and Sullivan, 2003) aspects can also be created to represent behavioural relationships, however Eos always selects advice execution associated with a *target* object.

The `associate(f, t)` method that binds the objects in the association, is automatically defined when the `perobjects` modifier is used. The modifier also defines a `delete()` method that revokes an association. In contrast to perobject aspects in AspectJ, the creation and destruction of association-aspects instances is explicit.

8.3.2. The contract hierarchy

As can be seen from the declaration of `FTContract` in Listing 8-1 and from the class model in in Figure 8-5 above, concrete contracts inherit from the abstract performative contracts (such as Supervisor-Subordinate, Peer-Peer, Auditor-Auditee etc.). In the example, `FTContract` inherits its control behaviour from the Supervisor-Subordinate contract `SuperSub`. Performative contracts in turn inherit from the abstract aspect class Contract.aj

8.3.2.1. The root aspect – Contract.aj

This top-level aspect contains pointcut declarations that define types of method invocation. These types are the CCA (Control-Communication Act) abstract methods we introduced in Chapter 6. In the current implementation of the ROAD framework, the CCA of a method is indicated by a prefix on the method name[7]. For example, an invocation to perform some action is represented by a `do_` prefix, as in `do_makeThingy()`. The named pointcut to trap calls to this method would therefore be

```
 pointcut doIt() : call(public * FnRole+.do_*(..));
```

which matches a public method whose name starts with "do_", that returns any type, has any parameters, and is in a class that inherits from the functional role abstract class `FnRole`. Listing 8-2 below shows the Contract aspect definition with some of its CCA pointcuts. Using the OR operator || to combine pointcuts, it is possible to define a named compound pointcut `allCCAs()` that represents all valid organisational communication between roles. This allows us to prevent any object that is not of type `FnRole` calling CCA methods, as we will discuss below (Listing 8-5).

The abstract aspect `Contract.aj` is also responsible for maintaining references to the roles bound to the contract and providing the data structures storing the terms of the contract. Consequently, it implements methods for adding, removing and getting parties and terms. Note that `Contract.aj` is a standard AspectJ aspect (not an

[7] Since AspectJ v1.5 it has been possible to define pointcuts on Java annotations. This is a less intrusive way to indicate the control semantics of a method, rather than altering the method name. However, CCAs have not yet implemented this way as the Association-Aspect compiler is based on AspectJ v1.2.

association aspect) as it only defines some general pointcuts, data structures and methods.

```
public abstract aspect Contract
{
    // FnRole+ is any subclass of FnRole
    pointcut doIt() : call(public * FnRole+.do_*(..));
    pointcut setGoal() : call(public void FnRole+.setG_*(..));
    pointcut inform() : call(public void FnRole+.inf_*(..));
    pointcut query() : call(public * FnRole+.qry_*(..));
    ...
    pointcut allCCAs() :  doIt() || setGoal() || inform() || query() ||
          resAlloc() || resReq();
    ...

    protected Vector terms = new Vector(); //The list of terms for
contract
    protected FnRole partyA;
    protected FnRole partyB;
    protected Organiser organiser;

    ...
```

Listing 8-2: The top-level abstract aspect Contract.aj

8.3.2.2. Abstract performative contracts

In the ROAD framework, it is the performative-level of the contract hierarchy that does most of the work at runtime. Performative contracts such as `SuperSub` extend the aspect `Contract` by defining the terms of the contract at the level of abstract messages. They take the CCA patterns (pointcuts) defined in the super aspect `Contract.aj`, and combine them with abstract pointcuts that indicate the direction of communication.

These directional pointcuts are called `aToB` and `bToA` to indicate communication sent from party A to B, and B to A respectively. Performative aspects are association-aspects, taking the objects (in our case abstract interface references to the role instances) that are bound in the contract as arguments to the `perobjects` modifier. This mechanism allows us to create pointcuts for the role instances that have been associated in the contract. For example the abstract pointcut `aToB` is defined in the performative contract as shown in Listing 8-3 below

```
protected abstract pointcut aToB(Supervisor sup, Subordinate sub);
```

This abstract pointcut is made concrete in the concrete contract (e.g. `FTContract)` as follows:

```
protected pointcut aToB(Supervisor a, Subordinate b): this(a) &&
                                      target(b) && associated(a, b);
```

where `a` and `b` are functional roles that implement, respectively, the `Supervisor` and `Subordinate` interfaces. This pointcut specifies that the invocation is made by from party `a` to party `b`; i.e. the `Supervisor a` `(this(a))` and to target `Subordinate b` (`target(b)`), and that a and b are associated in a contract (`associated(a,b)`[8]). In this way we can create pointcuts that define contract terms, such as `a1` in Listing 8-3, that contain both a CCA pattern *and* a direction of invocation for the role in the contract. In other words, we can control the type of thing that can be said by one party (role) to another. The code below implements the example specification of 'who can say what' for a Supervisor-Subordinate contract as set out in Chapter 6 (Table 6-2). Note also that compound pointcuts such as `a0` can be created that represent all the types of communication one party can say to the other.

```
public abstract aspect SuperSub extends Contract perobjects(Supervisor,
Subordinate)
{
  protected abstract pointcut aToB(Supervisor sup, Subordinate sub);
  protected abstract pointcut bToA(Supervisor sup, Subordinate sub);

  //Supervisor invocations - Subordinate obligations
  pointcut a1(Supervisor sup, Subordinate sub): doIt() && aToB(sup, sub);
  pointcut a2(Supervisor sup, Subordinate sub): setGoal() && aToB(sup, sub);
  pointcut a3(Supervisor sup, Subordinate sub): inform()  && aToB(sup, sub);
  pointcut a4(Supervisor sup, Subordinate sub): query()  && aToB(sup, sub);
  pointcut a5(Supervisor sup, Subordinate sub): resAlloc() && aToB(sup, sub);
  pointcut a6(Supervisor sup, Subordinate sub): accept() && aToB(sup, sub);
  pointcut a7(Supervisor sup, Subordinate sub): reject() && aToB(sup, sub);

  //Subordinate invocations - Supervisor obligations
  pointcut b1(Supervisor sup, Subordinate sub): inform() && bToA(sup, sub);
  pointcut b2(Supervisor sup, Subordinate sub): query() && bToA(sup, sub);
  pointcut b3(Supervisor sup, Subordinate sub): resReq() && bToA(sup, sub);
  pointcut b4(Supervisor sup, Subordinate sub): accept() && aToB(sup, sub);

  //compound performatives
  //what party a can say to party b
  pointcut a0() : doIt() || setGoal() || inform() || query() || resAlloc() ||
          accept() || reject();
  // what party b can say to party a
  pointcut b0() : inform() || query() || resReq() || accept();
  //all the things they can say to each other
  pointcut c0() : a0() || b0();
  ...
```

Listing 8-3: Example of a performative contract between a Supervisor and Subordinate

These pointcuts only define the *interception points* in an interaction between two roles. These interception points allow us to define actions at those points using aspect *advice*. An *advice* is method-like code in an aspect that is executed when a *pointcut* is reached in the execution flow. AspectJ supports a number of types of advice including *before* advice (executed just before the *join point* is reached) and *after* advice (executed after returning normally or after returning with an error).

Figure 8-7 below[9] schematically shows how an instance of ROAD contract (a `SuperSub` contract instantiated by `ft1` of type `FTContract`) mediates a

[8] The `associated` pointcut is provided by the association-aspect compiler.
[9] This figure should be treated as expository rather than a literal sequence diagram. Aspect-Contracts are not just interceptor classes that sit in between roles, but rather its pointcuts are woven into the roles themselves.

synchronous interaction between the two functional roles (`Foreman f` and `ThingyMaker tm1`). As described above, contracts have *terms* that define obligated transactions between the parties. The contract intercepts method calls between parties bound by the contract, where the signatures of those methods are defined in a contract term. In the case below, DO CCA calls from `f` (party A) to `tm1` (party B) that start with a method name prefix `do_*` are intercepted. As shown in the figure, pointcuts can also be defined that prevent unauthorised method calls (any method call that is not specified in the contract) either between the parties to the contract, or from external entities.

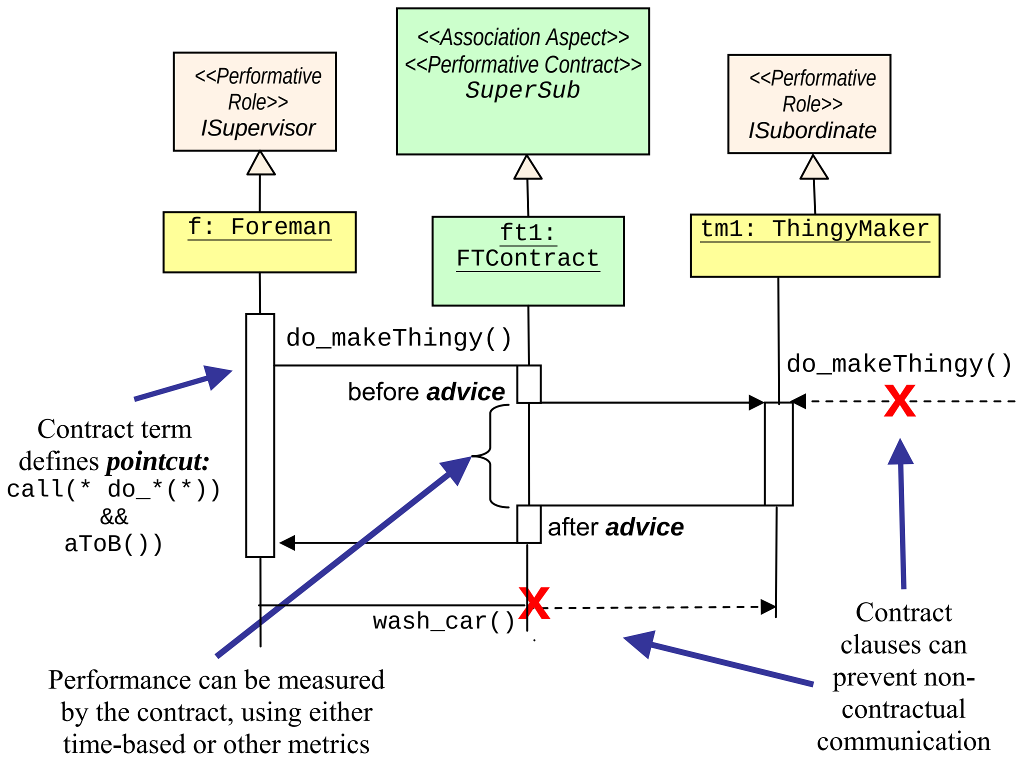

<<Performative Role>> ISupervisor
<<Association Aspect>> <<Performative Contract>> SuperSub
<<Performative Role>> ISubordinate
f: Foreman
ft1: FTContract
tm1: ThingyMaker
do_makeThingy()
before advice
do_makeThingy()
after advice
wash_car()
Contract term defines pointcut: call(* do_*(*)) && aToB())
Performance can be measured by the contract, using either time-based or other metrics
Contract clauses can prevent non-contractual communication

Figure 8-7: Synchronous transaction between roles under contractual control

The following code snippets from the `SuperSub` contract use the compound pointcuts `a1` and `b0` to prevent unauthorised communication between the parties. For example, the pointcut

```
aToB(a, b) && !a0( )
```

represents any communication that is from party A to B and is *not* of a CCA-type that A is allowed to invoke on B . As shown in Listing 8-4 below, a *before* advice is used to intercept the call and throw an exception if the pointcut is matched. The error string passed to the exception is built using AspectJ's `thisJoinPoint` primitive which gives reflective information on the execution context. For example, this pointcut and advice would intercept the Foreman-Supervisor's non-CCA request to the ThingyMaker-Subordinate to `wash_car()`as shown in Figure 8-7.

```
...
//prevent unathorised communication from a to b
before(Supervisor a, Subordinate b): aToB(a, b) && !a0()
{
   String s = "-X-> Unauthorised "+ thisJoinPoint.getKind() + " \"" +
              thisJoinPoint.getSignature() + "\" from "+
           thisJoinPoint.getThis().getClass().getName() + " to " +
           thisJoinPoint.getTarget().getClass().getName();
   throw new InvalidCCAException(s);
}
//prevent unathorised communication from b to a
before(Supervisor a, Subordinate b): bToA(a, b) && !b0()
{
   ...
   throw new InvalidCCAException(s);
}
...
```

Listing 8-4: Restricting communication between roles in a performative contract

Contracts can also prevent external parties (i.e. those objects that don't have a contract with the role) from interacting with the party. For example, no object other than a properly contracted role can invoke the ThingyMaker's `do_makeThingy()` method: not even another Supervisor-Foreman. The following *before* advice can be used to intercepts all CCA pattern calls from non-functional role objects (those that do inherit from `FnRole`):

```
before(Object a, FnRole b): allCCAs() && this(a) && target(b) {
    if (a != b){  //only call to other party, call within itself
             is ok
         if (!((FnRole) thisJoinPoint.getTarget())
                     .isContractedTo((FnRole)
             thisJoinPoint.getThis()))
                throw new InvalidCCAException(…);
}}
```

Listing 8-5: Intercepting unauthorised CCA calls from non-roles

Similarly, the advice in Listing 8-6 below intercepts *all* calls from non-contracted functional roles.

```
before(FnRole a, FnRole b): call(public * mContract.*.*(..)) &&
             this(a) && target(b) {...}
```

Listing 8-6: Intercepting unauthorised calls from uncontracted roles

As well as controlling interaction, abstract performative contracts define points in an interaction that allow performance to be measured — performance being a change of state from before to after a transaction. As discussed in Chapter 6, the points that are used to measure a transaction will vary depending on the synchronisation method (as in Figure 6-8). Our 'interaction diagram' in Figure 8-7 above shows a *synchronous* transaction between the parties. In a synchronous transaction, *before* and *after returning* advice associated with a contract term are used as the transaction is complete when the method call returns normally. If an error occurs during the transaction this is caught by an *after throwing* advice. If the transaction is *asynchronous*, the post-transaction state is measured when a method call matching a response signature is intercepted. As shown in class diagram Figure 8-5, asynchronous terms have a field that records the response signature. In general, a subordinate's performance is only measured on a DO CCA, i.e. when some action is invoked. For example, as shown in Listing 8-3 above, the term `a1` defines a DO CCA from a supervisor to a subordinate. The *before* advice will therefore be:

```
before(Supervisor a, Subordinate b): a1(a, b)
{
   beforeUpdate(thisJoinPoint.getSignature().getName(), Term.A_TO_B);
}
```

Listing 8-7: A performance measurement point before the start of a transaction

The performance measurement advice calls either a `beforeUpdate()` and `afterUpdate()` method (defined in Contract.aj) that take as argument the signature that identifies the term of the contract and the obligated party (`Term.A_TO_B` is a constant that indicates party B is obligated). This request to update performance is then passed to term, which in turn calls the `calculateUtility()` method of its utility function(s).

8.3.2.3. Concrete Contracts

All the above implementation details are part of the ROAD framework that is transparent to the application developer. The task for the application developer is to create concrete contracts by defining the contracts' specific terms and to implement the domain-specific utility functions. As discussed in Chapter 6, these utility functions measure the change in time or some other state. For example, the time elapsed between before and after advice can be used to calculate time-based metrics such as rate of production. Alternatively, some other utility function (such as cost) could be evaluated by accessing the execution context of the advice and measuring a change in state of the system or the environment.

Concrete contracts, such as the `FTContract` in Figure 8-5 above, allow us to define performance characteristics for each term of a contract object. Domain-specific characteristics are passed as parameters to the Term class constructor when the clause is created. These parameters include: a reference to the contract that the term belongs to; the method signature; the direction of the invocation (`aToB` or `bToA`); the synchronisation type; a response signature if the type is asynchronous; and a domain-specific utility function object that defines the performance metrics. The following code snippet from `FTContract` creates a term along with a utility function:

```
  ...
  Utility widgetUtility = new MakeThingyUtility(80, 45, 70, 100);
  //params:  absBreachThold, targetMeanMSec, avgBreachThold,
movAvWindow

  Term doMakeThingyTerm = new Term(this, "do_makeThingy", Term.A_TO_B,
              false, widgetUtility, SyncType.ASYNCHRONOUS,
              inf_thingyMade");
  ret = addTerm(doMakeThingyTerm, partyB);
  ...
```

Listing 8-8: Adding a term and its utility to a contract

The abstract `Utility` class has a `calculatePerformance()` method that can be overridden by a concrete domain-specific utility sub-class (such as `MakeThingyUtility` in Figure 8-5). Once the performance of a term is calculated, its state will be reported to the contract, if it is in breach or is underperforming. The contract, in turn, notifies its organiser of any underperformance.

In order for the functional-role classes to be able to work with performative contracts, they need to implement the empty interfaces that represent any applicable operational-performative roles such as `Supervisor` or `Subordinate` (as shown in Figure 8-2 above). They also need to extend the abstract functional role class `FnRole`. The creation of these dependencies does not require the alteration of the pre-existing functional roles but can be achieved by using an aspect with an *inter-type declaration.* Such declarations can create, at compile-time, the inheritance relationships and interfaces for functional roles. For instance, the following creates the inheritance relationships for the `ThingyMaker` and `DooverMaker` classes:

```
declare parents: {ThingyMaker || DooverMaker} extends FnRole
                                    implements Subordinate;
```

Some roles, such as Foreman, would implement both `Supervisor` and `Subordinate` interfaces.

8.3.3. Creating and revoking contract instances

Once we have created a contract type in the above form, the creation and revocation of contract instances at runtime is straightforward. The following code would be invoked by an Organiser to create then delete a contract of type `FTContract`:

```
//create a contract between f1 and tm1
FTContract ft1 = new FTContract(f1, tm1);
//create a contract between f1 and tm2
FTContract ft2 = new FTContract(f1, tm2);
//revoke the contract between f1 and tm1
ft1.delete(); //...
```

Listing 8-9: Creating and revoking a contract

8.3.4. Limitation of using association aspects

A limitation in the current implementation of the association-aspect compiler (Sakurai, Masuhara, Ubayashi et al., 2004) has become apparent during implementation. Different pointcuts within the same aspect generalisation hierarchy (such those in Figure 8-5) cannot match the same join point. This limitation prevents the specialisation of contract clauses using pointcuts: in other words, a functional contract cannot add extra advice to that which is already defined in the performative contract by means of defining its own pointcuts. Instead the advice in performative contract must invoke abstract methods that are over-ridden in the functional contract.

8.4. Self-managed composites and organisers

As describe in Chapter 7, self-managed composites are role-players that are themselves composed of roles. Figure 8-8 below is an implementation view of the concepts presented in Figure 7-1.

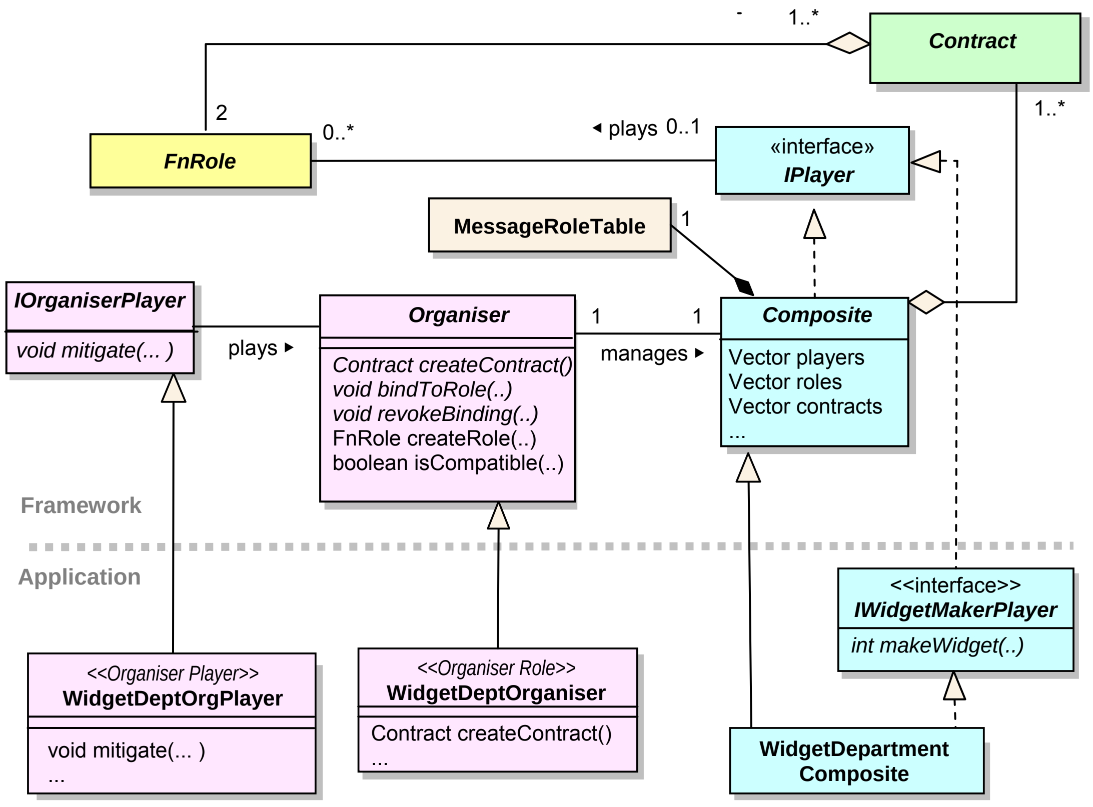


Figure 8-8: Composites, Organisers and associated classes

A composite has lists of its roles, players and contracts. These lists are maintained by the composite's organiser as it reconfigures the composite's structure.

As discussed in Chapter 7 (Figure 7-5), a composite has two interfaces: a functional interface over which operational process interactions flow, and a management interface of the non-functional requirements and performance data flows.

As a composite performs no domain function by itself, any functional messages must be delegated to a role. This delegation is implemented using a MessageRoleTable. The details of the MessageRoleTable are the same as in the FnRole as described in Section 8.2.2, and are not repeated here. A composite can be viewed as an inside-out role (or vice-versa). Whereas a role MessageRoleTable delegates *outgoing* messages from its player to externally contracted roles, a composite delegates *incoming* messages to its internal roles. As a composite is a role player, all outgoing messages are passed to the role it plays.

The management interface of a composite is the same as the external interface of its composite organiser role. (As the relationship between an organiser and its composite is always one-to-one, these concepts could be implemented as the one class.) In ROAD's current state of implementation, this interface is entirely domain-specific. A potential area of future work would be to define a generic management interface over which NFRs of various types could flow (similar to that proposed for the management of Web services (OASIS, 2005)).

As shown in Figure 8-8 above, organiser roles contain the methods for restructuring the composite. These methods such as `createRole()`, `createContract()` etc., maintain the composite's representation of itself up-to-date (in particular, the contracts, player bindings, and MessageRoleTables in its composites and roles). Every organiser role has a mutual reference to an organiser player which implements the IOrganiserPlayer interface. This interface defines a single abstract method `mitigate()`, which is called when any underperformance of the composite's contract is detected. This method is implemented in the concrete domain-specific Organiser Player (WidgetDeptOrgPlayer in Figure 8-8). As such, the ROAD framework does not specify the mitigation logic but a domain-specific version of a decision making process such as that illustrated in Figure 7-4 would be implemented.

8.5. State of framework implementation and further work

The current "proof-of-concept" implementation of the framework provides a set of abstract classes that enables the creation of adaptive organisational structures that can dynamically respond to changes in both non-functional requirements, and to changes in the performance of the players. The current framework supports both synchronous and asynchronous transactions in a 'push' mode of organisation. The ROAD concepts described in Part II of this thesis describe a broad vision of adaptive software, and further work is needed to fully develop the framework. Below is a list, in no particular order, of potential further developments of the framework.

- Mechanisms for the aggregation of non-functional requirements from contracts into role 'position descriptions'. Organiser players (which, being domain-specific, are outside the scope of the ROAD framework) could then be developed to use these descriptions to inform appropriate player discovery, and to negotiate appropriate external SLAs with external players.
- Support for the protocol and general clauses described conceptually in Chapter 6 have not yet been implemented in the framework. Protocol clauses require the development of state machines in the contracts that can represent and track sequences of transactions (Jin and Han, 2005).
- Transaction state machines that can handle timeouts and retries are yet to be developed.
- It may be useful to develop support for deferred-synchronous transactions.
- Pull-modes of process control are not yet supported.
- All communication between organisers is currently domain-specific. It may be worth developing and supporting a generic management language in the framework. This language could express concepts similar to those proposed in the Management-of-Web-Services (WSDM-MoWS) standard (OASIS, 2005).
- Additional types of role-player adaptor could be developed. Currently the framework only supports players developed in standard Java, or as Web services with a WSDL interface. A truly heterogeneous framework would support a wide range of technology such as RPC, CORBA and possibly agent communication languages such as FIPA-ACL.
- The version of association-aspects used to implement contracts in the framework only supports the compile-time weaving of the contract term pointcuts into the functional code. The limitation of compile-time weaving is that, while new contract *instances* can be added to an application at runtime, new contract *types* cannot be created on the fly. This prevents runtime *functional* (as opposed to non-functional) recomposition *within* a composite. However, as composites are players, new composites with different internal functional configurations can always be swapped at runtime. Recent developments, such as load-time weaving, support more dynamic deployment of aspects and could be used to dynamically create new types of contract.
- As discussed in Chapter 5, the maintenance of domain state during the player transition and reconfiguration is an outstanding issue. This needs to be addressed on both conceptual and implementation levels.

- Also discussed in Chapter 5, the semantics of the role-player interface will also vary depending on the degree of autonomy assigned to the role by the organisation (and the capability of the player). In the current implementation of ROAD, we have only implemented players with 'no autonomy' (as shown in Figure 5-5), and leave the implementation of other types of player to future work.
- A text-based declarative description (e.g. in XML) of contracts and composites would enable the dynamic creation of classes of these types. Such descriptions would be integral to the development of a ROADmaker tool, as we will discuss in the next section.

Beyond the above issues concerning further generic framework development, there are issues that need to be addressed in a domain-specific way. For example, the greater the uncertainty of the environment of the adaptive system, the greater will be the capability required of the organiser player(s). Although these domain-specific players are not strictly part of the framework, they will need to be developed in parallel with the framework proper, if the full potential of adaptive ROAD organisations is to be realised.

8.6. Towards tool support for developing organisational structures

Adding adaptive behaviour to an application increases its complexity and complicates the programmer's task. In its current state of development, the ROAD framework provides a set of abstract classes that can be extended by the application developer to create adaptive organisational structures (as in Figure 8-2 above). While the framework facilitates reuse, and hides much of the complexity of the adaptive mechanisms, to be a practical proposition software developers will need tool support – a 'ROADmaker' – to handle the added complexity.

Roles, contracts and composites provide an articulated[10] organisational structure over which functional messages flow between players. As discussed in Section 8.2.1 of this chapter, there is a dependency between the interfaces of players, roles and the contracts in a role structure. The aggregation of contract terms that bind the role to other roles, defines the organisation-side interfaces of the role, and those interfaces need to be re-expressed as an interface between the role and its player. The compatibility between all these interfaces must be maintained.

[10] A structure with 'joints' in it. These joints provide degrees of indirection which then have to be managed.

In conventional object-oriented or component programming, the required and provided interfaces between two communicating components must match. If component A has an interface `X` that *requires* certain methods be implemented in B and *provides* certain method implementations that B expects, then component B needs a complementary interface `X′` as shown in Figure 8-9 below.

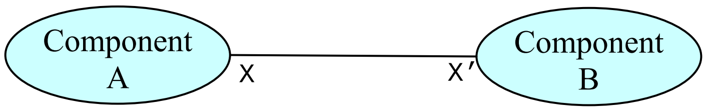


Figure 8-9: Interface dependencies between two communicating components / objects

Likewise in ROAD, when Player A communicates with Player B over the role structure, there still must be a basic compatibility of interfaces between the players, even if there is some transformation of the syntax or protocols of the communication. Figure 8-10 below shows the interdependencies between interfaces of roles and their players in a simple binary relationship, ignoring any alteration to the format of messages that may occur within the roles or their adaptors (see Section 8.2.4). If a player is a self-managed composite, such as Player B, then there must also be compatibility between the external interface of the composite `X′` and the interfaces of the internal roles to which the composite delegates messages (Roles D and E in the Figure, such that $X' = \Sigma(x'_1, x'_2)$).

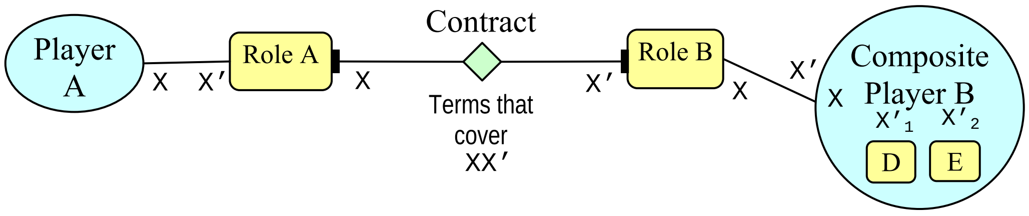


Figure 8-10: Interface interdependencies between roles, players and contracts

It follows that creating such a role structure involves a lot of redundant information in the expression of the interfaces. If the interfaces of the roles and players, as well as the contract terms, are all coded separately, then the application programming task becomes complex and possibly prone to error. To make the ROAD approach practical from a software development perspective, tool support will be needed to ensure consistency, and to reduce the amount of redundant coding needed (the interfaces should only need to be written once). A preliminary design for a ROADmaker Eclipse plugin has been developed that presents a graphical representation of the composite under-development. This tool would allow the programmer to create a composite from pre-existing interface definitions of objects / components / services. Alternatively, the programmer may want to design an organisation from scratch and generate code stubs / interface definitions for the players. While the development of such a tool is beyond the scope of this thesis, the nature of the dependencies in a ROAD role structure suggests a broad approach that could be taken in the development of such a tool. This approach involves dividing the domain-specific constructs that extend the ROAD framework into two categories.

The first category covers constructs that define the role structure. These structural constructs include roles (functional and organiser), contracts, adaptors and composites. At the programmatic level, these constructs can all be viewed as entities that define or re-express interfaces. A declarative programming approach is appropriate for defining such role structure constructs. This is because they do not, in themselves, define any procedure (or more precisely, their procedures and data structures are generic and thus derived from the framework). A data model of these structural constructs and their interdependencies could be developed, and then used as a schema for a programming construct database that is manipulated by the tool. The declarative elements of this model are as follows:

- Concrete Contracts consist of contract terms that can be declaratively defined. This is already the case in the current ROAD implementation, in the sense that terms are fully described by parameter values passed to a term constructor. These values could be expressed, say, in an XML document. The construction of the contract instance itself merely requires a declaration of references to its parties (the functional role instances).
- Functional Roles can be derived from the contract terms. All domain specific information in a role is either contained in its dynamically updatable RoleMessageTable (updated by the Organiser), or declaratively defined (e.g. its name). All role functionality is either inherited (e.g. message queuing), or provided by external entities (e.g. adaptors and allocators). Roles, therefore, should be able to be automatically generated.
- Composites perform a mirror routing function to the functional role (Figure 7-2). As with roles, this is achieved using a dynamically updatable RoleMessageTable. All the data structures in a composite are generic[11] and inherited from the framework. Maintenance of these structures is performed by the organiser. No non-declarative information is therefore needed to create a composite.

[11] This is the case in the current implementation. However, as discussed in Chapter 7.2, an alternative approach would be to store domain state in the composite.

- Role-player adaptors are dependant on the technologies used to implement the role and player that inter-connect. As such, they can be automatically generated, perhaps subject to some manual intervention needed to resolve mismatches at the protocol or syntax level.
- Domain-specific organiser roles inherit all their reconfiguration functionality from the framework. They also define methods to make contracts of given types. These methods are identical, other than their type information, and thus could be generated automatically using reflective mechanisms provided by Java, if we know the types to be contracted.

The second category of constructs includes those that “do the work”. These are: the functional players that perform the domain process; the organiser players that decide what reconfiguration and regulation is necessary; the allocators that decide on work allocation to roles of the same type; and the utility functions that calculate the performance of the role-players. These constructs need a non-declarative style of programming.

8.7. Summary

The ROAD Framework described in this chapter provides a set of abstract classes that the application developer can extend to create an ontogenically adaptive application. These classes hide much of the complexity of the adaptive mechanisms that allow the application to cope with changing requirements and with changing environments. A prototype implementation has been implemented in Java with an extension to AspectJ called ‘association aspects’. In ROAD, instances of these aspects are used to implement contracts that are the connectors in the organisational structure, that control interactions and measure performance over that structure. Using these aspects, an abstract hierarchy of contract types can be created. This allows the definition of reusable abstract performative contracts that can define common patterns of interactions between role types. The implementation of role and composite abstract classes is also described. Like contracts, these classes provide generic functionality needed by concrete roles and composites, such as message routing and queuing.

The current implementation of the ROAD Framework supports both synchronous and asynchronous interaction in a ‘push’ mode of organisation. There is a number of extensions that could be made to the current implementation. In particular, a graphical programming tool could be developed to assist the application developer declaratively create organisational structure and check their consistency. The next chapter (Chapter

9) describes an application built on this framework and shows its adaptive capability. The expressiveness and efficiency of the framework are discussed in Chapter 11.

9

A Test Application

This chapter describes the implementation of a test application based on the ROAD framework we described in the previous chapter. The application[1] demonstrates the adaptive behaviour that the ROAD framework facilitates. The test application is based on our running example of a Widget Making department. The application demonstrates the following capabilities of the ROAD framework.

- Role creation
- Contract creation and deletion
- Adding terms to contracts
- Control of unauthorised communication
- Performance measurement
- Player selection based on performance
- Work allocation
- Binding roles to heterogeneous players

In order to demonstrate cause and effect in the test application, we have constructed the composite 'from the outside' by using a test harness. This allows us to simply show the application's output relative to its input. In production code the composite would create itself (create its roles and contracts) based on some declarative description of its structure.

This rest of this chapter is structured as follows. The first section provides an overview of the classes used in the test harness and application. Section 9.2 shows how

[1] This chapter is based on the technical report by Pham, Colman and Han (Pham, Colman and Han, 2005) "The Implementation of Message Synchronisation, Queuing and Allocation in the ROAD Framework".

the composite is created using roles, players and contracts, and how the framework prevents unauthorised communication. Section 9.3 shows how the organiser of the Widget Department attempts to mitigate underperformance by reconfiguring the composite. Section 9.4 then shows how a ROAD application can work with heterogeneous players – in this case Java objects and Web services.

9.1. Overview of the test application

As shown in Figure 9-1 below, the test harness uses a package of domain specific organisational classes (Widget organisation package) based on the ROAD framework, to create an instance of a WidgetDepartment composite that plays a WidgetMaker role. The WidgetMaker role receives widget orders from the ProductionManager whose player is driven by the user input via the test harness. Foreman and ThingyMaker roles are added to the WidgetDepartment composite, along with players who can play different roles (i.e. implement the appropriate interface). These players have different performance characteristics, which are simulated by putting various delays in their methods that process functional requests.

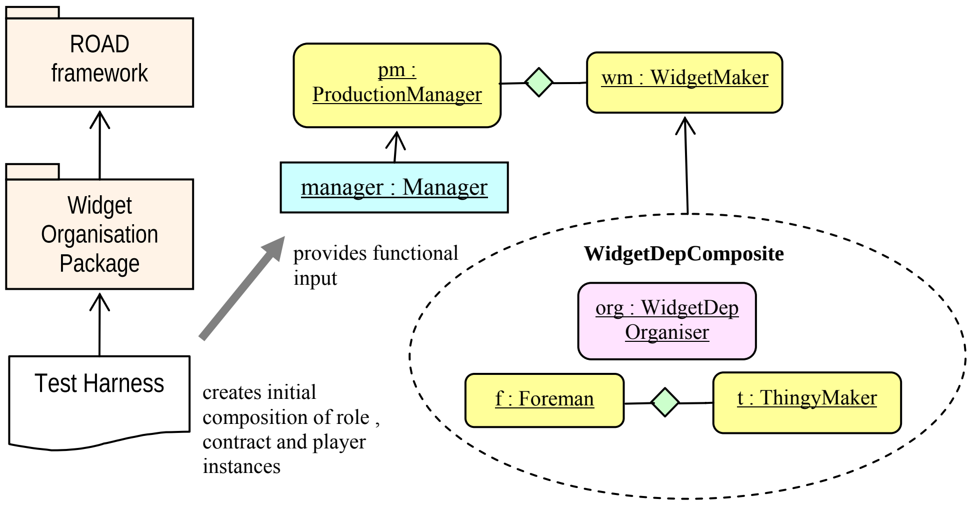

ROAD framework
Widget Organisation Package
Test Harness
pm : ProductionManager
wm : WidgetMaker
manager : Manager
provides functional input
creates initial composition of role , contract and player instances
WidgetDepComposite
org : WidgetDep Organiser
f : Foreman
t : ThingyMaker

Figure 9-1: Test harness creates initial configuration based on Widget Organisation Package and simulates functional load via Production Manager

The ability of ROAD to intercept and prevent unauthorised interactions is then demonstrated by attempting to invoke methods in the WidgetMaker and ThingyMaker, prior to creating the appropriate contracts. The appropriate contracts are then created between the ProductionManager and the WidgetDepartment, and between Foreman and the ThingyMaker, respectively. Once the composite is properly constructed, an order to make widgets is invoked. The output of the program then shows how the

Linh Duy Pham was primarily responsible for the writing of the original report and the associated test application code.

WidgetDepartmentOrganiser reconfigures the composite as it attempts to meet the terms of its contract. The subsequent sections discuss the test harness's input, and the subsequent output in more detail. Figure 9-2 below shows the main classes from the Widget Organisation package that are used in the test harness.

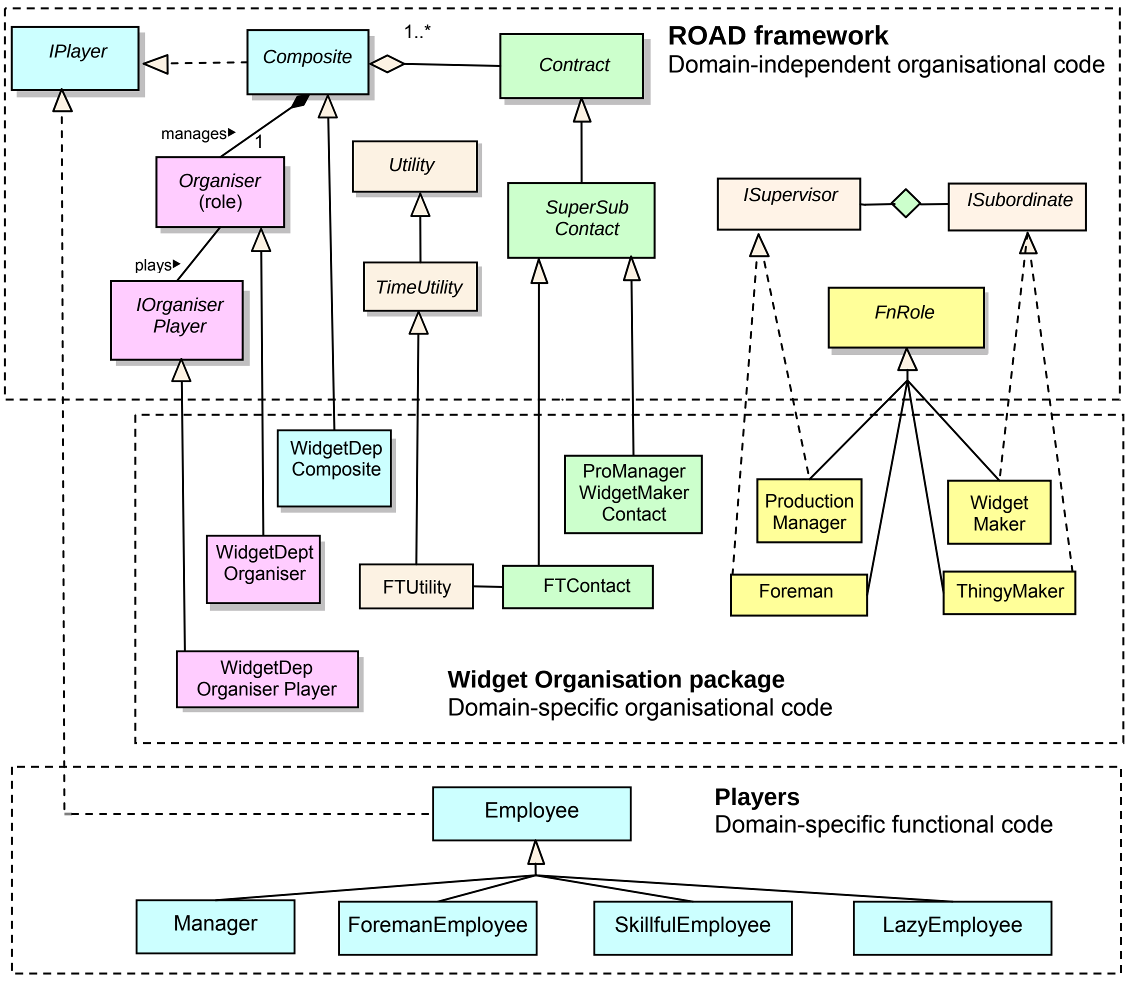


Figure 9-2: Overview of the main classes of the application used by the test harness

9.2. Composite construction and controlling communication

The following discussion refers to code from the test harness and to the subsequent output as found in Appendix A. References to line numbers from the test harness code will be prefixed with an "i" (e.g. i21), while references to output line are referenced with an "o" (e.g. o89).

9.2.1. Creating the composite, roles and players

The test harness script begins (i25 to i47 as shown in Listing 9-1) by creating instances of roles, players, a composite and organiser from the classes defined in the Widget Organisation package. SkillfulEmployee objects are players that can be assigned to both

Foreman role and ThingyMaker role. Players of type LazyEmployee can only be assigned to ThingyMaker role. Inside the WidgetDepComposite object, there are three ThingyMaker players: badThingyMakerPlayer (of type LazyEmployee), and a foremanPlayer, goodThingyMakerPlayer (of type SkillfulEmployee).

```
ProductionManager pm = new ProductionManager("Production Manager");
WidgetMaker wm = new WidgetMaker("Widget Maker");

//player for ProductionManager
Manager manager = new Manager("Manager");

// Organiser setup
WidgetDepOrganiserPlayer orgPlayer = new WidgetDepOrganiserPlayer();
Organiser org = new WidgetDepOrganiser(new WidgetDepRoleFactory());
org.setPlayer(orgPlayer);

// Player for WidgetMaker --> create the WidgetDepComposite
Composite widgetDepComposite = new WidgetDepComposite();
org.setComposite(widgetDepComposite);

//create ThingyMaker and Foreman
ThingyMaker t = new ThingyMaker("Thingy Maker");
Foreman f = new Foreman("Foreman");

//create other players
SkillfulEmployee foremanPlayer = new SkillfulEmployee("Foreman/Thingy Player");
SkillfulEmployee goodThingyMakerPlayer = new SkillfulEmployee("Skillful ThingyMaker");
LazyEmployee badThingyMakerPlayer = new LazyEmployee("Lazy Thingy Maker");
```

Listing 9-1: Creating the entities for the test application

The entities created are shown in Figure 9-3 below.

9.2.2. Adding roles to composite and binding players

Once the entities have been created, the roles and players are added to the composite (i54 to i80). Initially, one Foreman role, and one ThingyMaker role are added. foremanPlayer (of type SkillfulEmployee) is then bound to Foreman f, badThingyMakerPlayer (of type LazyEmployee) is bound to ThingyMaker t. The initial players of the roles can be changed later during execution time if the players do not meet the performance requirements. The time taken to make a thingy is arbitrarily chosen and it is implemented by Thread.sleep() method. A SkillfulEmployee object produces a thingy in 20 ms. Whereas a LazyEmployee object produces a thingy in 10 ms, however after each thingy is made, its performance delay is increased by 20 ms (i.e. 30 ms then 50 ms, and so on). The cap performance delay is 100 ms, after which it remains unchanged. Listing

9-2 below shows the lines from the test harness used to add roles and players to the composite and binding the players to the roles.

```
54    // add Roles and Players to composite
55    widgetDepComposite.addRole(f);
56    widgetDepComposite.addRole(t);
58    widgetDepComposite.addPlayer(foremanPlayer);
59    widgetDepComposite.addPlayer(goodThingyMakerPlayer);
60    widgetDepComposite.addPlayer(badThingyMakerPlayer);
61    // Setup Initial Players
62    try
63    {
65            pm.setPlayer(manager);                  //ProductionManager
67            wm.setPlayer(widgetDepComposite);       //WidgetMaker
70            f.setPlayer(foremanPlayer);             //Foreman and ThingyMaker
73            t.setPlayer(badThingyMakerPlayer);
74    }
```

Listing 9-2: Adding entities to composite and binding players to them

In this case the players have been added manually to the composite as the Widget Department Organiser player has no player discovery capability. Figure 9-3 below shows the initial configuration of the WidgetDepComposite and the composite's relationship. At this stage (line i80) the contracts between the functional roles have *not* yet been created. To simplify the code the ProductionManager and WidgetMaker have been created in the context of the test harness rather than created by another composite.

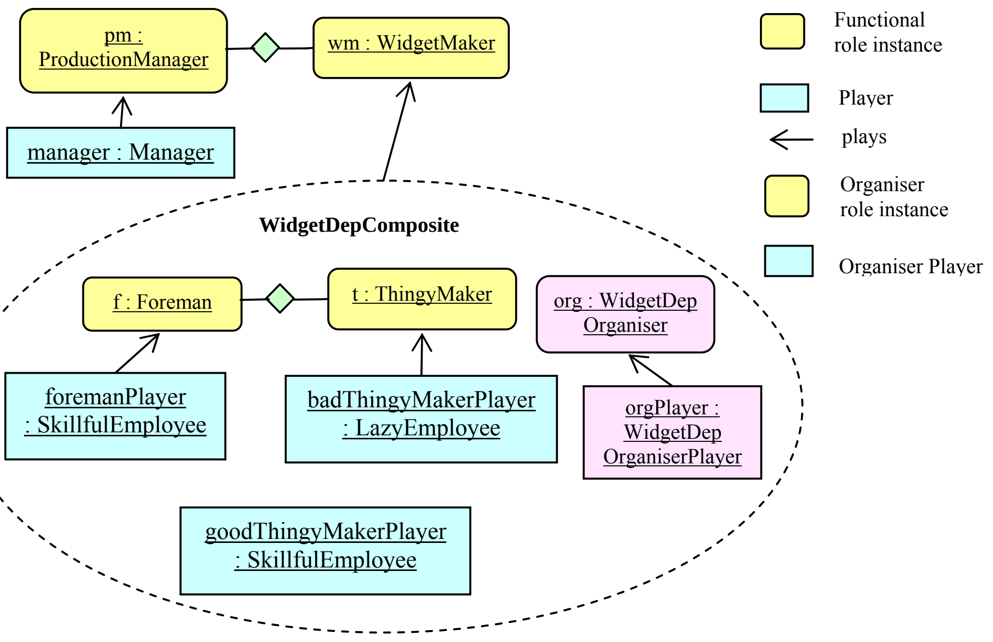


Figure 9-3: Initial configuration WidgetDepComposite Instance

9.2.3. Preventing unauthorised communication

The next section of the test harness demonstrates how non-contracted communication is prevented. As the contract between ProductionManager and WidgetMaker has not been created yet, the framework will prevent the invalid CCA between two uncontracted roles. The order of widgets is placed by calling the method do_placeOrderWidgets() of ProductionManager (i82 to i94). The method prompts the user to enter a number. After taking the input of the number of widgets required from the user, the ProductionManager sends request to WidgetMaker. The invalid CCA between uncontracted ProductionManager and WidgetMaker is prevented (o13).

Similarly, when the Foreman attempts to send messages to ThingyMaker for thingies to be made, these messages are blocked since, at this stage (i107), the Foreman and ThingyMaker are uncontracted roles. Listing 9-3 below (o26 to o28) shows the status messages generated by the performative contract.

```
23    ---- Before contract between Foreman and ThingyMaker is created ----
24    TEST: Should have error non contracted between Foreman and ThingyMaker
25    To user: Enter number of widgets required: 15
26    ---> before a1 do AtoB : call(void widgets.WidgetMaker.do_makeWidget(int)) – Calculate Start time.
27    ---> after a0 error: call(void widgets.WidgetMaker.do_makeWidget(int))
28    X--X CCA call from uncontracted functional role: call(void widgets.ThingyMaker.do_makeThingy())
```

Listing 9-3: Output showing unauthorised call being prevented

9.2.4. Creating contracts

The contract ProManagerWidgetMakerContract is created between ProductionManager and WidgetMaker (i99).

```
ProManagerWidgetMakerContract contract = new ProManagerWidgetMakerContract(pm,
                                         wm);
```

Likewise, a contract between the Foreman and the ThingyMaker is created (i117 reflected o31 to o35) by calling the method

```
org.createContract(f, t);
```

The organiser determines the appropriate contract type based on the type of the roles passed as parameters. In this case, the terms of the contracts have been hard-coded in to the contract. A functionally valid structure has now been created. When an order for widgets is made pm.do_placeOrderWidgets(); (i125), the functional messages can now flow through the organisational structure.

9.3. Adaptive behaviour

The input from the test harness ends once a widget order has been invoked on the functionally viable structure. Even though the structure is functionally valid, it is not necessarily viable in terms of the non-functional requirements expressed in the various contracts. The rest of the output (o38 to o281) shows how the composite copes with the changing performance of its ThingyMaker players as the order of 15 widgets is processed. As, in our simplified example, a widget is made of only one thingy we only show thingy in the output. The requests from Foreman to ThingyMaker to make thingies are asynchronous requests. For the order of 15 widgets from the ProductionManager player, the Foreman will place 15 requests to the ThingyMaker with each request being for 1 thingy.

The composite's organiser (orgPlayer : WidgetDepOrganiserPlayer) uses a basic strategy when faced with underperformance in a contract. Firstly, it looks to find the player with the best performance from those it has available. In the case of the test application the organiser has no discovery mechanism for players – it only has available those that have been previously made known to it. The organiser then compares the best player's claimed performance with the existing player's actual performance. If the new player's performance is better it will use it. If there is no better player, the organiser will then create another role and attach another player to it so that there are now multiple role-players performing the function. (In more sophisticated organiser-players more advanced strategies could, of course, be used.)

There are two threads running in the background, one is the main thread, the other one is the thread inside the ThingyMaker role processing the asynchronous messages. The main thread pushes a message into ThingyMaker role's message queue (o47). The message processing thread then pops the message out and processes it. The first thingy is made by badThingyMakerPlayer in 10 ms Line (o50). The elapsed time is slightly more than 10 ms (o53 shows 15 ms) because of the delay while the message is inside the queue. The performance of the badThingyMakerPlayer progressively degrades (o64 to o81) until it reaches 94 ms and it is in breach of the contract requirements. The organiser then replaces the badThingyMakerPlayer by a better player (o89). In this case, the better player is either foremanPlayer or goodThingyMakerPlayer as they both have the performance of 20 ms. The choice is random and foremanPlayer is chosen.

The configuration of the WidgetDepComposite instance is thus changed dynamically. The foremanPlayer is now assigned to both Foreman and ThingyMaker roles.

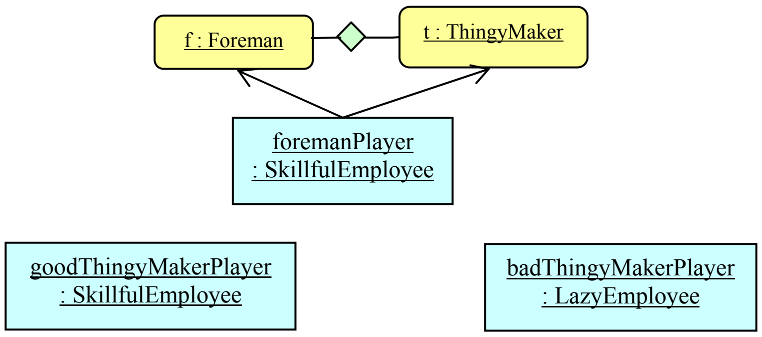

f : Foreman
t : ThingyMaker
foremanPlayer : SkillfulEmployee
goodThingyMakerPlayer : SkillfulEmployee
badThingyMakerPlayer : LazyEmployee

Figure 9-4: The Configuration of WidgetDepComposite Is Changed Dynamically. The Foreman player takes on the ThingyMaker role.

The next thingy is nominally made in 20ms by this newly assigned ThingyMaker player (o98), although in this case queuing overhead results in a measurement performance of 93 ms (o101). Since we have just changed the player, the average performance of this new player is calculated based only on the new player's performance. This performance once again is in breach of the contract requirements. However, there is no better player. The organiser detects that the composite has unused resources; in this case, goodThingyMakerPlayer and badThingyMakerPlayer. To minimise the change in the configuration, it tries to utilise one player at the time. To utilise a ThingyMaker player, it has to create a new ThingyMaker role, and create a contract between this new ThingyMaker role and the existing Foreman. It then binds the best available player (in this case, goodThingyMakerPlayer with a performance of 20 ms) to the newly created ThingyMaker role.

The new configuration is now shown in Figure 9-5.

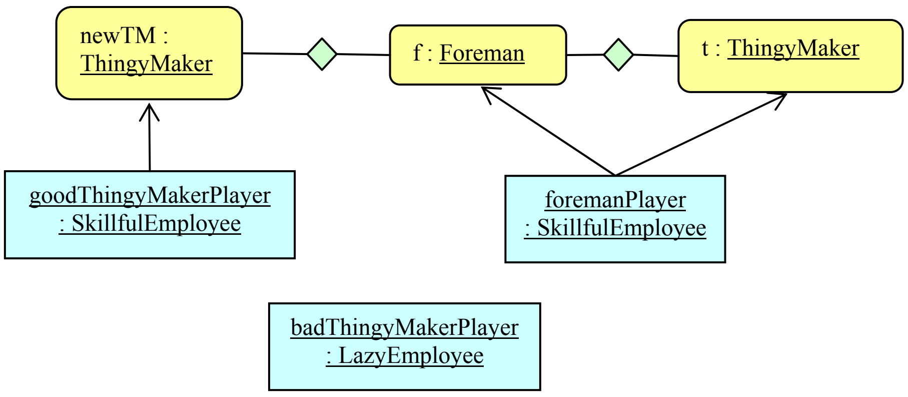

newTM : ThingyMaker
f : Foreman
t : ThingyMaker
goodThingyMakerPlayer : SkillfulEmployee
foremanPlayer : SkillfulEmployee
badThingyMakerPlayer : LazyEmployee

Figure 9-5: Re-configuration of the role structure – another ThingyMaker role is created

Since there is a message processing thread inside each ThingyMaker role instance, there are three threads running at this moment. One is the main thread, and the other two are threads inside two ThingyMaker role instances. The main thread is putting messages from Foreman to the two ThingyMaker instances, and the other two threads are processing the messages. The work allocator provided by the framework determines which role receives the message (in this case a round-robin allocation scheme has been designated in the Foreman's MessageRoleTable when the FTContract is created). The output at this point is quite hard to follow because these threads are executing independently.

As the goodThingyMakerPlayer participates in creating thingies (o153) the elapsed time and average are calculated separately for each of the players. At the end of the output, the program does not terminate, since the main thread finishes but the other two threads in ThingyMaker instances are still running and waiting for new messages in their queues.

9.4. Heterogeneous players

The ability of the ROAD framework to cope with heterogeneous players is tested using a separate test harness. This test harness is substantially the same as the one in Appendix A except in this case a wsThingyPlayer of type ThingyPlayerWebServiceProxy is employed. A Web service to make thingies is created and run on a server. The initial composite configuration is shown in Figure 9-6 below.

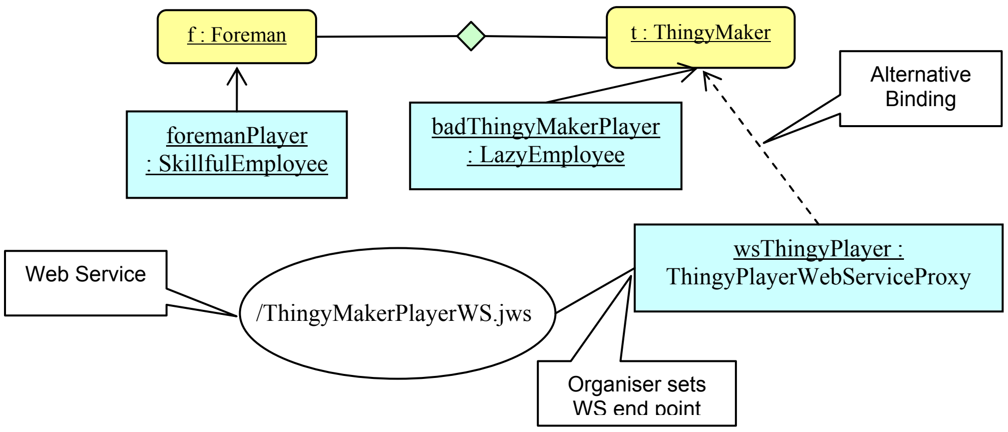

f : Foreman
t : ThingyMaker
Alternative Binding
foremanPlayer : SkillfulEmployee
badThingyMakerPlayer : LazyEmployee
Web Service
/ThingyMakerPlayerWS.jws
wsThingyPlayer : ThingyPlayerWebServiceProxy
Organiser sets WS end point

Figure 9-6: Using a Web service player

In the case of a Web service player, the organiser has two bindings to control: the binding between the role and the player (in this case a proxy for the Web service), as well as the binding between the proxy and the Web service itself[2]. Using the Apache Axis platform to provide the implementation of SOAP (Apache Web Services Project, 2006), the proxy-service binding merely involves setting the endpoint URL of the service as follows:

[2] This is slightly different approach to that described in Section 5.5 as there is no adaptor attached to the role. Rather the proxy is the player and there is an extra binding.

```
   String endPoint = "http://[some
url:port]/axis/ThingyMakerPlayerWS.jws";
   wsThingyPlayer.bindToNewEndPointService( new URL(endPoint) );
```

In terms of adaptive behaviour the Web service proxy wsThingyPlayer appears like any other player. Performance of the service is measured in exactly the same way, and the same mitigation strategies can be used. The only difference with the standard case is that at runtime, the organiser has an additional level of indirection available to it by controlling the Web service end point.

9.5. Summary

The above test application demonstrates the basic adaptive behaviour of the ROAD framework, including the creation of new roles and contracts, as well as the selection of players. In addition the test application shows how unauthorised communication is controlled and how ROAD handles heterogeneous players. The adaptive behaviour demonstrated in the test is in response to changes in the performance of players, rather than changes in the requirements set in the contracts. However, as the adaptive behaviour is triggered in response to the *difference* between requirement and performance, perturbation caused by changing requirements is substantially similar to the case tested here. The above test example also focuses on the adaptive behaviour within a single composite as it plays a role in an enclosing composite (in this case simulated by the test harness). The next chapter presents a design case study that shows how a ROAD application would be created that involves multiple levels of composites. The performance of the ROAD framework is then discussed in Chapter 11.

10

A Design Case Study in Service Oriented Computing

This chapter demonstrates the application of ROAD to a domain that contrasts with our running example, the Widget department. This case study shows how a Book Broking service can be designed to assist a book purchaser (a large library) effectively acquire books from various book sellers via the Web services those sellers provide.

This case study highlights a number of different characteristics of ROAD. These include:

- How ROAD can be applied to Service Oriented Computing (e.g. Web services), in particular the mediation between changing requirements and the changing provision of services.
- The use of ROAD in an information system rather than a manufacturing control system.
- Types of abstract performative contract suitable to commercial inter-organisational contracts (as distinct from intra-organisational contracts such as supervisor-subordinate, peer-peer, etc.).
- The use of contracts which govern 'long-lived' transactions.
- The ability to represent 'virtual enterprises'[1] using ROAD. Transactions in these enterprises are not entirely ad hoc, but occur in the context of dynamic longer term relationships.

[1] For example, as covered by IFIP working group 5.5 (Cooperation Infrastructure for Virtual Enterprises and Electronic Business - "COVE") http://www.ifip.org/ or the Society of Collaborative Networks http://www.uninova.pt/~socolnet/

- The creation of protocol clauses from sequences of transactions.
- The measurement of non-time based utility in the performance of a contract. A number of utility functions described in the case study are informational representations of the state of the domain (e.g. financial and physical domain) rather than a measurement of the time of the interaction itself.
- The aggregation of compound utility functions to determine the over-all utility of a relationship.
- The design of composites so that role abstractions in a composite are always kept at the one level of abstraction. In ROAD, roles are not decomposed into other roles within a composite; rather, roles are always played by loosely coupled players. It is these composite players that decompose the function into the (internal) roles, and these players can always be separated from the (external) role they play. Highly adaptive systems can thus be created, because the decomposition (not just the configuration) can always be changed at runtime.

The chapter is structured as follows. The first section describes the business context of the case study, and defines some relevant system requirements. The second section presents a high-level ROAD design of the system where composites and roles are assigned responsibilities in fulfilling the system's requirements. Section 10.3 decomposes the composites identified in the high-level design into roles. Section 10.4 specifies some of the contracts between these roles. The fifth section discusses the management interface for a composite in terms of the various types of utility objects that are passed over that interface. Section 10.6 illustrates the adaptive behaviour of the system, and then 10.7 discusses the case study as an example of 'application-specific' service-oriented middleware. The chapter is briefly summarised in Section 10.8.

10.1. Context and system requirements

A large institutional library purchases many books from many suppliers. This purchasing is currently the responsibility of the Acquisition Department (AD) that works in collaboration with other individuals/departments (Librarians, Accounts, etc.). The same book can often be sourced from multiple suppliers.

Because the library is such a large purchaser of books, its suppliers have an interest in providing a high level of service to the library (competitive prices, quick turn-around etc). Likewise the library has an interest in maintaining good working relationships with key suppliers. It generally tries to limit the number of suppliers in order to keep administrative overheads down. Ordering through a limited number of suppliers also enables the library to place larger orders with those suppliers and thus negotiate quantity discounts. As distinct from an individual purchase transaction, the terms of a library-supplier relationship are more long-lived. The properties of a relationship between the library and a supplier often vary. These variables (we will call them "terms of trade") could include:

- payment on invoice or statement
- time from order to delivery (average, range, deviation, maximum, ...)
- terms of payment (prepayment, on invoice, 30 days, 60 days, ...)
- discount on RRP, if any
- who pays freight
- reliability of supplier with respect to previous deliveries
- product range supplier has available
- duration of relationship
- value of trade with supplier
- reputation of supplier (particularly if prepayment is required)
- and so on ...

The library has set the values of its desired outcomes of these set of variables (e.g. low price, payment 30 days on statement, freight free-into-store, delivery not more than 2 weeks unless otherwise specified, and so on). The Acquisitions Department currently use these policies to negotiate terms of trade with individual suppliers.

In recent years the choice of suppliers available to the library has greatly expanded (e.g. the library can now order books directly from overseas). This has resulted in increased price competition from suppliers. Also, the ordering and payment for books can now be transacted on-line. To take advantage of these changed market conditions, the library would like to partially automate the process of search, selection and payment for its books. In order to obtain competitive prices, the library plans to develop an in-house automated broking service ("the Broker").

This service will provide a Web Service interface to the library, and will purchase books automatically using the suppliers' Web services interfaces[2]. Many book suppliers provide automated Web service interfaces for book search, quoting, ordering and payment. The protocols for these transactions and the terms of trade vary between each supplier, and the Broker needs to be able to work with all of them. The responsibility of the Broker is to source the books needed by the library with optimal terms of trade for

[2] Some book sellers already provide Web services. For example, the XMethods site http://xmethods.net/ve2/index.po lists a number of book seller Web services e.g. http://www.abundanttech.com/webservices/bnprice/bnprice.wsdl http://majordojo.com/amazon_query/amazon_query.wsdl

the library, while maintaining strong relationships with suppliers. The Broker therefore has to assess the value of each transaction to the library (not just the price). It also has to ensure that the library and supplier have matching payment and delivery protocols.

In assessing the library’s optimal terms of trade, the Broker has to abide by the purchasing policies and preferences set by the library. These preferences often have to be traded-off. For example, the Broker can source books either locally or from overseas. While overseas books are often cheaper, delivery times are typically longer. Methods of payment also vary between local and overseas suppliers. Most (not all) local suppliers are happy to supply books on invoice as the library is a reliable payer. Overseas suppliers invariably want payment up front. The library therefore sets a purchasing policy (general terms-of-trade) that provides rules to the Broker so that it can trade-off these variables. For example, it instructs the Broker that it prefers to pay on statement or on invoice, yet it is prepared to pay upfront if the total cost saving is greater than 15% of the book(s) price, and the supplier has a good reputation.

The advantage for the Library using the Broker is that the Broker hides many of the details of relationships with suppliers, and automates the supplier selection process. Once its broking service is established, the Library hopes to be able to sell the service to other libraries. This would have the dual benefit of providing fee-for-service income to the Library, and of increasing its buying power through an increased volume of orders. The Library therefore requires that the Broker’s architecture be easily extensible so that it can cope with multiple libraries. The Library, Broker and the suppliers can be thought of as forming a ‘virtual enterprise’ that creates dynamic business relationships between its entities.

10.2. High-level ROAD-based design

10.2.1. Single library

To design a role-based system that meets the requirements set out in the above scenario, we decompose the system into composites that play a role with respect to the system. These composites are themselves made up of interacting roles, all of which are at a similar level of abstraction, and within a single domain of control (e.g. have a single owner). The Library Book purchasing system can be viewed as a virtual enterprise that consists of three composites and a number of other players. The composites are the Library itself, the Acquisitions Department composite that plays the Acquisitions role within the library, and a Book Broker composite that supplies the Acquisitions Department with books. An overview of the design is illustrated in Figure 10-1 below. This preliminary design does not include payment or delivery mechanisms – we assume payment is somehow made and recorded at some point in the transaction process, and that the book is somehow delivered to the library after purchase.

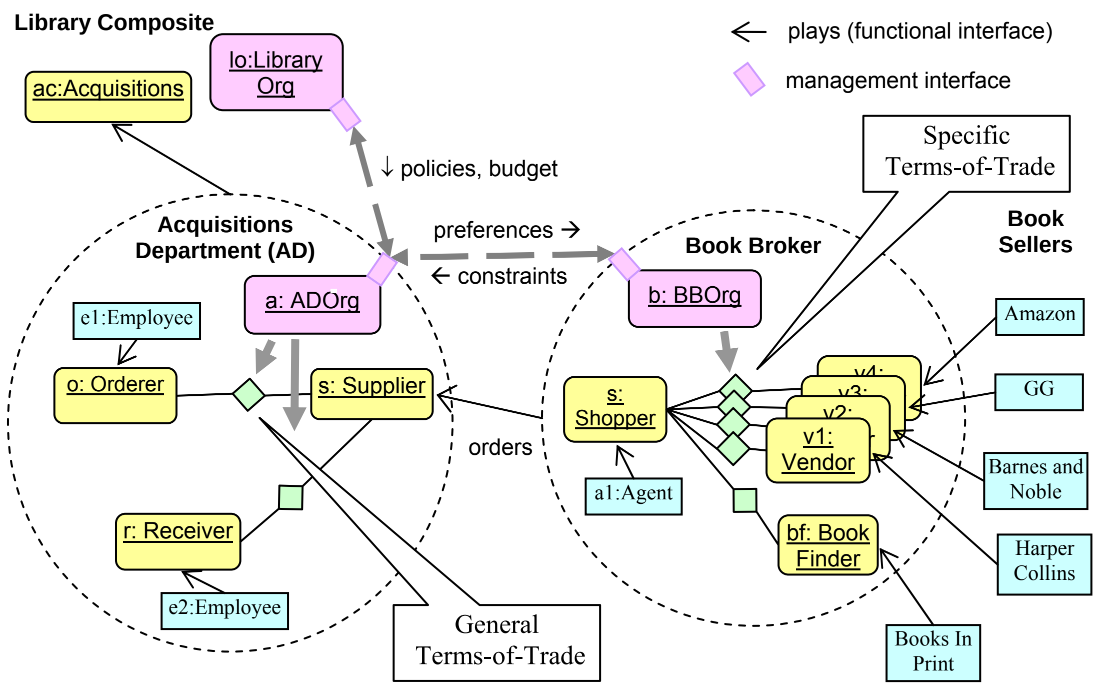


Figure 10-1: Library Book Broker Virtual Enterprise – example instantiation within a single library composite

Note that the structure of the Acquisitions Department and the Book Broker are both largely generic. The roles and their relationships in these composites are typical of purchasing and broking functions. As such, these composites could be specialised instances of more generic composites (e.g. the Book Broker is a special case of a Broker).

Also note that in the Broker composite each Vendor is represented by a different role, rather than there being a single Vendor role which is attached to different Book Seller players at different times[3]. This is because the agreed specific terms of trade and history of transactions need to be persistent for each Shopper-Vendor relationship, i.e. contract[4]. As shown in Figure 10-1, the organiser of the Book Broker (BBOrg) receives preferences and constraints (general terms-of-trade) from the organiser of the

[3] This is not just an implementation issue, because if there is only one role for which players are selected then conceptually selection is the responsibility of the Broker Organiser role. Otherwise choosing a supplier is a work allocation task that is performed by the Shopper.
[4] In implementing these contracts it would be sensible to store their state in a common database so that the Broker organiser BBOrg can run queries across contracts. As transactions pass through the contract, it calls update queries on the database

Acquisitions Department (ADOrg). BBOrg interprets these general terms-of-trade into specific terms-of-trade appropriate to each vendor, and stores them in the Shopper-Vendor contracts. The roles within these composites are discussed in Section 10.3.

10.2.2. Multiple libraries

Extending the design of the Broker to handle multiple library clients is straightforward, as shown in Figure 10-2 below. We will call this composite a Broking Service. A Broking Service consists of a number of Client-Broker relationships, there being one broker for each client. The contracts between a Client and Broker store a copy of the general terms-of-trade for that Client. For each library that uses the Broking Service, a separate Client role is created that acts as proxy for that library. All functional interactions with a library pass through its Client role. Each Broker role instance is played by a BookBroker composite (as in Figure 10-1 above). The organiser of the Broking Service composite passes the general terms of trade for a particular library client of the organiser of the contracted BookBroker composite (not shown in the figure). That BookBroker organiser then creates the specific terms-of-trade between Shopper and Vendor roles inside its composite, storing these requirements in the appropriate Shopper-Vendor contract. Note that this extension of the design can be achieved without making any changes to the previously defined Library and BookBroker composites.

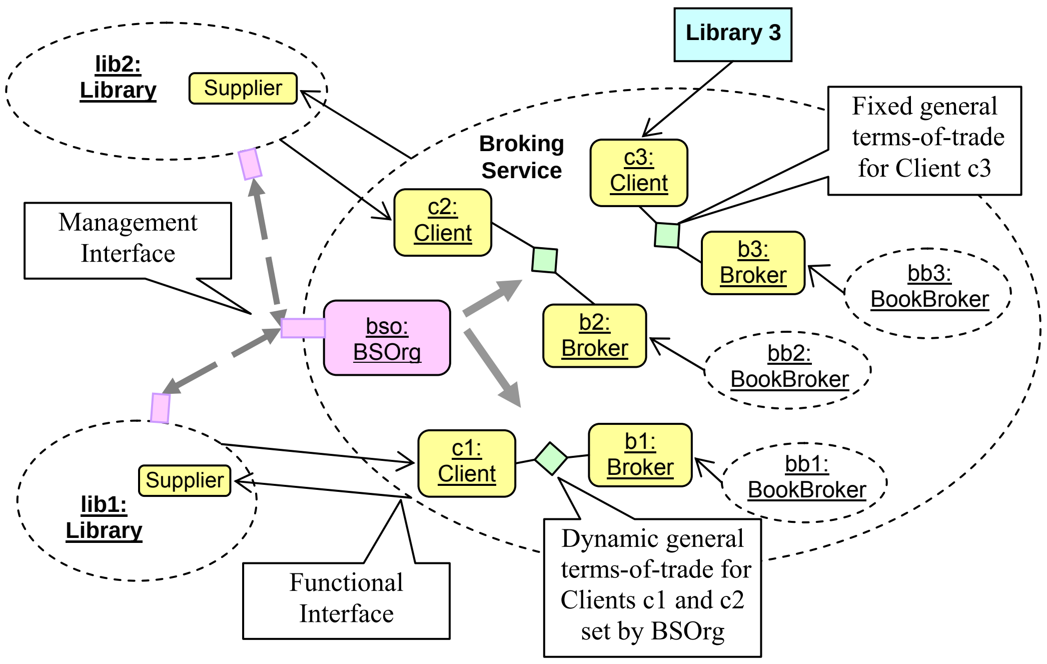


Figure 10-2: Independent Broking Service instance with multiple library clients

The Broking Service instance illustrated above is associated with two types of library. The first type of library is a ROAD composite, as instanced by Libraries lib1 and lib2 that play Client roles c1 and c2 respectively. As both the Library and the Broking Service are ROAD composites, they both present a management interface over which changing NFRs and performance data flow between their respective organisers. In this case, the organisers within the libraries' acquisition departments send the Broking Service NFRs relating to their general terms-of-trade. The organiser of the Broking Service (BSOrg) stores these NFRs in the respective Client-Broker contracts, and transmits them to the organiser (a BBOrg) of the appropriate Book Broking composite (not shown). We will discuss this management interface in more detail in Section 10.5.

The second type of library (Library 3) is *not* a ROAD composite. It has a functionally compatible Web service interface through which it interacts with the Broking Service, but it has no management interface or organiser. Although Library 3 can still order books, there is therefore no mechanism for dynamically changing the general terms-of-trade (NFRs) between Library 3 (playing the c3 Client role) and its Broker,. These NFRs would need to set statically in advance (or through some form of supervisory control via BSOrg). As a further enhancement of its services, the Broking Service could offer a public Web service interface for individual (non-institutional) customers who do not require a long term relationship with book suppliers. The organiser would dynamically create ad hoc client and broker roles, and a broker player. The contracts in the composites would have fixed terms-of-trade (e.g. pay up front).

Note that in the above design, a composite Library and a Broking Service both play a role in each other's organisation, as indicated in Figure 10-2 by the *plays* relationship arrows between the composites. The Library's AD Supplier role is played by a Broking Service, and the Book Service's Client role is played by a Library. The implication of this is that all functional messages between the composites pass through these respective roles. Such a structure would suit composites in different organisational domains (e.g. with different owners), because each composite has an internal role that is a proxy for the other composite. This will assist the decoupling of the Broker service in the event of it being 'spun-off' as a separate business entity.

10.3. Decomposition of composites

The responsibilities of the AD, Book Broker and Broking Service composites with respect to the book purchasing virtual enterprise are defined by the roles they play, i.e. the Library.Acquisitions role and the AD.Supplier role. A Book Broker and a Broking Service are both Supplier role players, and would look functionally identical to a composite Library. The only difference between them is that the Broking Service can handle multiple Library clients. In this section, we will examine in more detail the single library composition shown in Figure 10-1, and decompose the responsibilities of the AD and Book Broker composites into their internal roles.

10.3.1. Responsibilities of the Acquisitions Department

The Library's Acquisition Department (AD) is responsible for acquiring books on behalf of the Library. An AD composite models the relationships between roles involved in the acquisition process: Orderer; the Receiver; and the Supplier. In the instantiation of the system shown Figure 10-1 the Orderer and Receiver roles are played by library employees (using an appropriate UI) and the Supplier role is played by a Book Broker (or Broking Service) composite.

Orderer role position description

- Receive requests for books from the Library. (These requests are delegated from the composite's MessageRoleTable.)
- Prioritise orders
- Request quote from Supplier
- Check sufficient funds available
- Create order (with order number) and forward orders to the Supplier (played by the Broker)
- Follow protocol for the purchase of books as defined in its contract
- Request and receive funds from the Library. Keep budget actual balance updated.

Supplier role position description

- Receive requests for quotes and orders from Orderer
- Provide quotes
- Fill orders
- Receive notification from the Receiver of receipt of the ordered items. (The Supplier role is played by the Broker which calculates performance data (e.g. delivery time) of particular orders because it involves the physical delivery of items.)

Receiver role position description

- Physically check in-coming books against delivery and order documentation
- Inform Supplier of receipt of order.
- Notify Supplier of any discrepancies between what was ordered and what was received.

AD Organiser role (ADOrg) position description

- Sets the general terms of trade (NFRs) in the Orderer-Supplier contract for purchasing books from the polices set by the library.
- Notifies the organiser of the Broker (BBOrg) of those terms. These NFRs can either be firm constraints or preferences.
- Binds players to roles (in the case of the AD composite these relationships would be relatively stable).
- Handles exception passed to it from the BBOrg, e.g. unable to find supplier of book given constraints.

10.3.2. Responsibilities of the Book Broker

Externally the BookBroker composite needs to meet the requirements of the Supplier role in the AD composite. Internally the Broker is decomposed into Shopper, Vendor and BookFinder roles. The Shopper role receives the AD's book orders by delegation from the Broker composite. The Vendor roles are proxies for book supplier Web services, and the BookFinder role is proxy for a book search service (played by the BooksInPrint Web service).

Shopper role position description

- Send queries to the BookFinder role to find which suppliers have copies of the book(s) that match the library's order.
- Find which potential suppliers have existing contracts (agreed terms-of-trade) with the library.
- If none of the suppliers are currently contracted with the library client, inform the organiser so a new contract can be negotiated
- Send requests for quotes to the contracted Vendors who stock the book
- Evaluate quotes to find the most favourable. This is a special case of the work allocation function discussed in Chapter 8.3.2.
- If there is an acceptable offer order the book(s) according to the protocol defined in the contract.
- Send notification of the order (Order No, Price) to the composite's role allocation table (this is then passed to the Library via its AD.Supplier role).
- If no acceptable offers are forthcoming, inform the composite.

Vendor role position description

The Vendor role is a proxy for an external book supplier. Book suppliers use various protocols but all have the following responsibilities:

- Provide a response to following queries
    - whether a book is in stock and how many copies are available
    - quote prices on an order including transport charges
- Fill a book order based on a quote and arrange delivery
- If required the supplier may also be required to invoice/provide statements to the client directly
- Accept payment

BookFinder role position description

- Accept queries about the availability of book(s)
- Return name of book vendors who have stock of those books

Broker Organiser role (BBOrg) position description

We assume that the BBOrg player has a discovery mechanism for Web service book suppliers, and that these services provide adequate service descriptions (including use-protocols). If appropriate the BBOrg player would also negotiate specific terms of trade with external book suppliers consistent with the Library's general terms-of-trade.

- Create new Vendor roles, and Shopper-Vendor contracts.
- Get general terms-of-trade preferences from the organiser of the enclosing composite (ADOrg or BSOrg)
- Set the terms of the contracts between the Shopper and specific Vendors based on specific client-supplier relationship information (e.g. required method of payment).
- Revise contracts if general terms-of-trade are revised.

10.4. Contracts

As pointed out in Chapter 8, role definitions are aggregations of the contract terms that bind them, and players must be compatible with these role definitions. This section will define the contracts in the AD and Broker composites. We will firstly define some abstract performative contracts that define the interaction patterns, and then specify the concrete contracts.

10.4.1. Abstract performative contracts

Our design identifies two types of abstract performative contract: a Buyer-Seller contract and an Information Peer-Peer. For example, the interaction in a Client-Broker concrete contract can be generalised as a BuyerSeller performative contract as shown in Figure 10-3 below.

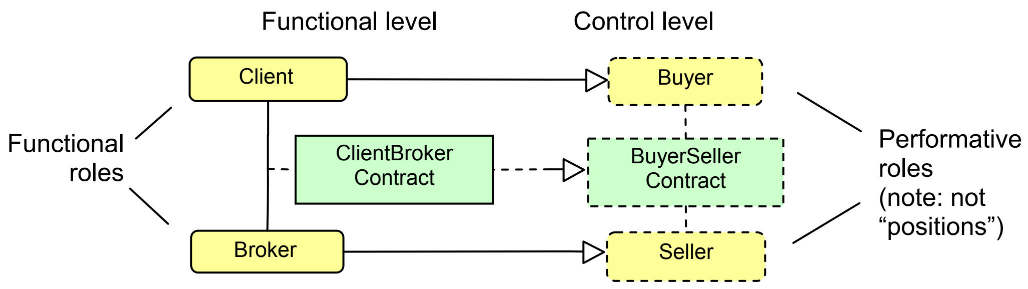


Figure 10-3: The Client-Broker relationship generalised as a Buyer-Seller contract

All concrete contracts in the design can be inherited from the Buyer-Seller and Information Peer-Peer abstract contracts as shown in the table below.

Table 10-1: Inheritance from common Performative Contracts

Performative Contract	Functional Contract
Buyer-Seller	Orderer-Supplier
	Shopper-Vendor
	Client-Broker
Information Peer-peer	Supplier-Receiver
	Shopper-BookFinder

We next define these two performative contracts in the form previously discussed in Chapter 6.

Buyer-Seller Performative Contract

The Buyer-Seller performative contract (Figure 10-4) defines the general form of communication between a Buyer and Seller. All transactions are asynchronous, that is, in the form of an initiated CCA and a matching response (e.g. GetQuote, Quote). Named transactions that can be initiated by the Party A (the Buyer) start with the letter "a" (e.g. `a2`), while Party B's (the Seller) transactions start with "b". As described in Chapter 6, protocol clauses are sequences of transactions. The example Buyer-Seller contract has a number of named protocols (`p1-p6`) that represent prototypical sequences of transactions that are permitted under the contract. These can be used to enforce appropriate business protocols. For example, the "pay up front" protocol `p1` consists of the transactions `a2,a3,b2`; i.e. (Order , OrderConfirm), (Pay-Receipt), (Deliver, Acknowledge). Protocols can also be formed from other protocols. For example, the "quoted pay up front" protocol `p4` consists of transaction `a1` (GetQuote, Quote) followed by protocol `p1`. A concrete contract that inherits from Buyer-Seller would have a subset of these protocol clauses activated; activations which, if necessary, changed at runtime. The final section of the contract is the 'performance measurement points'. These define the before and after points at which change of state can be measured to evaluate the performance of an obligated party.

Performative Contract Name: Buyer-Seller
Parties:
Party A: Buyer
Party B: Seller

CCAs:
(shorthand CCA code in brackets)
GetQuote (gq), Order (o), Pay (p), Acknowledge (ack)
Quote (q), OrderConfirm (oc), Deliver (d), Invoice (inv)

Transactions:
Party A initiated terms
a1: GetQuote-Quote
a2: Order-OrderConfirm
a3: Pay-Acknowledge

Party B initiated terms
b1: Invoice-Pay
b2: Deliver-Acknowledge

Protocols:
(Protocols are sequences of transaction terms – all Party A initiated)

```
p1: a2, a3, b2        //pay up front
p2: a2, b2, a3        //pay on delivery
p3: a2, b2, b1        //pay on invoice
p4: a1, p1            //quoted pay up front
p5: a1, p2            //quoted pay on delivery
p6: a1, p3            //quoted pay on invoice
```

Performance measurement points:
a1: price( ); discountRRP( ); ... *i.e. before and after a1*
b2: daysToPay( ) *i.e. before and after b2*
a2, b2: deliveryTime( ) *i.e. before a2 and after b2*
p1: deliveryTime( ) *i.e. before and after p1*

Figure 10-4: Buyer-Seller Performative Contract

Information Peer-Peer Performative Contract

An Information Peer-Peer contract shown in Figure 10-5 allows the contracted parties to query each other, or to provide unsolicited information. This is similar to the Peer-Peer contract in our Widget making application, except that one peer cannot invoke a DO CCA in the other. Note that, as this contract defines a simple request-response transaction, no multi-transaction protocols are defined. Additionally, as the contract involves information exchange only, no performance measure points are defined (although it would be possible to define one for timeliness of the response).

```
 Performative Contract Name: InformationPeerPeer

 Parties:
          Party A: Peer1
          Party B: Peer2
 CCAs:
          (shorthand CCA code in brackets)
          Query (qry), Inform (inf)
Transactions:
          Party A initiated terms
                    a1: Query, Inform
                    a2: Inform
          Party B initiated terms
                    b1: Query, Inform
                    b2: Inform
Protocols:
          NA
Performance measurement points for utilities:
NA
```

Figure 10-5: Information Peer-Peer Performative Contract

10.4.2. Concrete functional contracts

This section specifies some example concrete contracts from the Book Broker application. A concrete contract defines the specific messages each party can send to the other, and associates those invocations with a CCA. As previously, the convention is that the CCA of the method invocation is indicated as a prefix (e.g. **gq_**). As pointed out above in section 10.4.1, the Orderer-Supplier, Shopper-Vendor and Client-Broker concrete contracts all inherit from the abstract Buyer-Seller contract. These concrete contracts all serve the same function in different composites, indeed they could be all implemented with the same concrete contract type, provided the respective interfaces in the roles (e.g. Supplier, Vendor) are the same. We will assume this is the case and will define a single concrete contract BookBuyer-Seller to cover all the above associations.

```
Name: BookBuyer-Seller contract extends Buyer-Seller

Orderer initiated
          gq_GetQuote(Items)
          o_OrderBooks(OrderNo, Items, MaxTimeToStore, QuoteID)
          p_PayOrder(PayID, OrderNo, InvoiceNo, Amount)
          ack_OrderReceived(OrderNo)
          ack_OrderReceived(OrderNo, Discrepancy)
Supplier inititated
          q_Quote(QuoteID, Items, Price)
          oc_OrderConfirmed(OrderNo, Price, SupplierID)
          oc_OrderConfirmed(OrderNo, False, Reason)  //order not confirmed
          d_DeliveryNote(DeliverID, OrderNo)
          i_Invoice(OrderID(s), Amount)
          ack_Receipt(InvNo, Amount)
          ack_Receipt(OrderNo, Amount)
```

Figure 10-6: BookBuyer-Seller Concrete Contract

Recall that the *terms* of a contract are created dynamically, including details of the signature(s), the synchronisation type, and any utility function used to measure the performance of the term. Contract clauses that cover any protocols to be followed or other general clauses are also dynamically added. In our example, utility functions associated with BookBuyer-Seller contract terms could measure Price, DeliveryTime, or any of the parameters associated with the general terms-of-trade listed in Section 10.1. Utility function classes (as ROAD is written in Java these functions have been implemented as classes) have methods for setting the expected value of the parameters (e.g. for a DeliveryTimeUtility these values might be TargetMeanDeliveryTime, BreachThresholdDeliveryTime, etc.). As well as storing the *required* performance, a utility object stores the *actual* performance based on the history of interactions between the contracted roles. Each utility class implements a calculateUtility( ) method that returns a value associated with the contract performance (e.g. InBreach, UnderPerforming, Performing). Protocol clause classes are also dynamically added to contracts. These classes maintain state machines that define acceptable sequences of terms (e.g. as defined in the Buyer-Seller performative contract in Figure 10-4).

Other concrete contracts would be similarly specified. For example the Supplier-Receiver contract is much simpler than the Buyer-Seller performative contract. It is only used to create a connection and prohibit unauthorised interaction, rather than measure performance. As shown in Figure 10-7, the concrete contract merely associates particular method invocations with CCAs defined in the InformationPeerPeer performative contract.

Name: **Supplier-Receiver contract extends InformationPeerPeer**

```
Receiver initiated
        inf_OrderReceived(OrderNo)
        inf_OrderReceived(OrderNo, Discrepency)
Supplier initiated
        qry_GetOrderStatus(OrderNo)
```

Figure 10-7: Supplier-Receiver Concrete Contract

10.5. Composite management interfaces

A composite player has to meet the performance requirements (NFRs) that are contained in the contract(s) that bind the role it plays. For example, the BookBroker composite plays the AD.Supplier role, and as such it needs to meet the NFRs that obligate the AD.Supplier role. The contracts that define the AD.Supplier role are the Orderer-Supplier contract and the Supplier-Receiver contracts. However, only the Orderer-Supplier contract has NFRs defined against its terms, therefore the

BookBroker composite only has to fulfil the requirements associated with that contract. As discussed in the previous section, these NFRs are defined in the AD's Orderer-Supplier contract as instances of utility classes attached to the contract's terms. Copies of these utility objects are passed by the organiser of the super composite (ADOrg) to the organiser of the player composite (BBOrg), as shown in Figure 10-8 below.

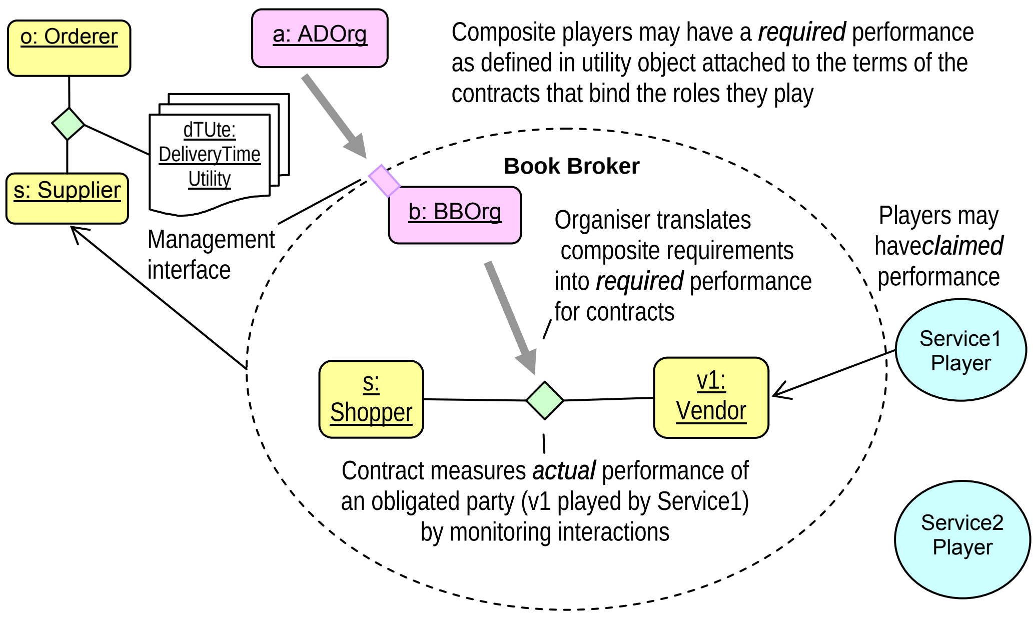

o: Orderer
a: ADOrg
Composite players may have a *required* performance as defined in utility object attached to the terms of the contracts that bind the roles they play
dTUte: DeliveryTime Utility
s: Supplier
Management interface
Book Broker
b: BBOrg
Organiser translates composite requirements into *required* performance for contracts
Players may have *claimed* performance
Service1 Player
s: Shopper
v1: Vendor
Contract measures *actual* performance of an obligated party (v1 played by Service1) by monitoring interactions
Service2 Player

Figure 10-8: Non-functional requirement and the measurement of their performance

Once the organiser of a composite is informed of NFRs in the form of utility objects, these composite-wide requirements (in our example, the general term-of-trade) need to be translated by the organiser into performance requirements for the utility objects associated with the contracts it controls (e.g. instances of the Shopper-Vendor Contracts). The organiser also uses this required performance information to compare the performance requirements of a role instance (e.g. v1) with any claimed performance of players that are candidates to play that role. If a role is bound by more than one contract, the organiser may need to aggregate NFRs into a consolidated position description for the role.

In our earlier example of a Widget Department, all utility was time-based; that is, performance could be measured numerically. However, not all NFRs are necessarily measured numerically. For example, preferences might be expressed for the type of protocol to be used. A NFR could consist of a list of protocols in preferential order. For example, the order of preference for the protocols (as defined in the Buyer-Seller performative contract) might be p6, p5, p4, while protocols p1, p2, p3 are unacceptable.

Utility objects may also be multi-variant, in that they contain a number of performance variables that need to be traded-off against each other. The trade-off may be based on rules contained in the calculateUtility( ) method of the utility object. For example, the library has a requirement that it is only prepared to deal with untrusted parties on the basis of Protocol p6 (pay on quoted invoice). Alternatively, a multi-variant value function (Keeney and Raiffa, 1976) could be used as the basis for evaluating particular trades or comparing the offers of particular vendors. Typically, such value functions reduce all attributes to a numeric value that allows these attributes to be compared. The values are then weighted to reflect their relative importance/preference in trading-off the attributes. The sum of the weighted values gives a single number that can be used to compare alternatives.

Table 10-2 below gives some examples of NFRs from the BookBroker. These include conditions for evaluating numeric and categorical variables, as well as single and multi-variant conditions.

Table 10-2: Examples of the evaluation of non-functional requirements

	Contractual Requirement	**Comparative Function**
Single Variable - numeric	Discount RRP > 5%	Minimum ($Price_A$, $Price_B$, $Price_C$, ...)
Single Variable - categorical	Protocol = (p6 || p5 || p4)	PreferredRank(p6, p5, p4)
Multi-variant	If Vendor.player.trusted = FALSE then Protocol = p6	$SomeUtilityFunction_{MAX}$ (Price, DeliveryTime, ...)

10.6. Adaptive behaviour

In this section we will describe the adaptive behaviour of the BookBroker composite. As described in Chapter 7, adaptive behaviour occurs *within* a ROAD composite and *across* composites. Adaptive behaviour within a composite involves the organiser following strategies for reconfiguration (creating/destroying roles, contracts and role-player bindings), and regulation (altering the terms of contracts). Adaptive behaviour across composites involves the transmission of NFRs between composites as described in the previous section, and the transmission of performance information in the case of capacity planning or composite underperformance.

The organiser of the BookBroker composite (BBOrg) first needs to establish relationships between its library client and various suppliers. The organiser receives NFRs representing the library's general terms-of-trade and needs to discover book suppliers with whom it can potential form contracts[5]. For each of the suppliers that meet the minimum standard for terms-of-trade, the organiser creates a Vendor role, binds the external service to that role (i.e. creates the Web service adaptor and sets the end-point), and creates an instance of a BookBuyer-Seller contract between the instance of the Vendor role and the Shopper role. Terms appropriate to the specific relationship (the specific terms-of-trade) are then written into each contract. Adaptation to change in requirements in this scenario could be triggered by either the client or a vendor wanting to change the terms of trade. If the organiser can find a mutual accommodation under the new requirements, the terms of the contract would be rewritten; if not the contract and role would be destroyed (and thus end the particular relationship to the supplier).

Once the BookBroker composite structure is established, and the terms of the contracts written, the composite can respond to book orders. One strategy the organiser BBOrg could employ would be to create a general utility function that ranks each of these vendors according to some comparative value function (as in Table 10-2) that weights each of the client library's preferences (discount, reputation, reliability payment terms, etc.) and calculates each supplier performance with respect to those weighted performance measures. This value function could then be used by the player of the Shopper role (the Agent `a1` in Figure 10-1) to decide where to place an order, given the list of suppliers (obtained by the external BookFinder service) who have the book(s). The Shopper may even break the order into a number of sub-orders, if the utility is greater. Once order(s) are placed and received, performance metrics (e.g. time to deliver, actual discount received, etc) can be updated in the appropriate Shopper-Vendor contract and fed back into the evaluation process for subsequent orders.

If there is no contracted supplier that can supply the book under existing terms of trade, then the organiser BBOrg is informed. The organiser may have additional strategies (e.g. search for new suppliers and create new contracts). If the order still cannot be filled, the problem is escalated back to the library's Acquisition Department organiser (ADOrg) who needs to decide if it wants to relax its general terms-of-trade for that order.

10.7. Discussion - ROAD as application-specific middleware

This case study highlights that a ROAD composite (*sans* services) can be viewed as service-oriented computing middleware (Colman, Pham, Han et al., 2006; Colman and

[5] We assume there are external mechanisms for service discovery, reputation assessment, and negotiation available to the organiser using WS standards such as UDDI, WS-Coordination and WS-Agreement and that suppliers also support these standards

Han, 2005). By using a common middleware, heterogeneous applications are able to communicate and collaborate. Middleware technologies also hide complexity and add value to these interactions. For example, taking a high-level Enterprise Application Integration (EAI) view, Figure 10-9 is a simple schema of heterogeneous applications that communicate via a conventional middleware layer that handles various properties related to their interaction (e.g. reliable messaging, logging, persistence).

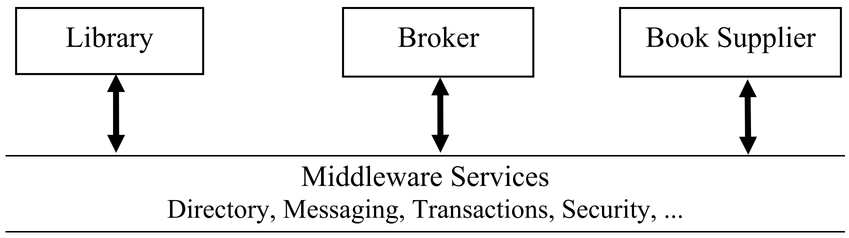


Figure 10-9: Conventional Middleware

In the more open environments typical of service-oriented computing, a number of shortcomings of the conventional view of middleware become apparent, particularly with cross-organisational Web services rather than conventional middleware integration within a single organisation. Cross-organisational composition of services involves issues such as lack of trust and asynchronous long-lived transactions, as well as deployment issues such as the location of the middleware in a distributed system (Alonso, Casati, Kuno, and Machiraju, 2004).

The other major shortcoming of the conventional middleware concept is the necessity for all the parties to an interaction to agree on, and use, interoperable standards. In the fast-changing world of Web services, although the basic technology for handling point-to-point interactions is well established and accepted, standards for handling more complex interactions such as WS-Coordination, WS-Agreement, OWL-S, and so on, are still evolving and are sometimes overlapping, depending on the domain and the originating standards body (e.g. (W3C, 2004), (OASIS, 2005), (Global Grid Forum, 2004), etc.). In order to build adaptable applications in this changing and uncertain technical context, it would be desirable for applications to be able to make use of heterogeneous standards, while not being bound to any one standard. Although we cannot do without standards, the challenge remains to create integrated applications that can make use of the heterogeneous middleware technologies, but that are not dependent on any particular technology. To use an analogy: programming languages like Java can run on heterogeneous operating systems by providing an independent layer (Bytecode running on the JVM) between the application code and the operating system. In order to preserve this 'Write Once, Run Anywhere' approach to software development, we need to avoid tightly coupling the application to the execution environment, whether that environment be an operating system or middleware.

ROAD can be viewed as an "application-specific" middleware. Application-specific middleware might sound like a contradiction of terms, because middleware standards and technologies are almost by definition generic. However, in application-specific middleware, like conventional middleware, all messages between the component services pass through a middleware layer. In addition, the application-specific middleware provides adaptive structures for the composition, control of service interactions, and the measurement of QoS of those interactions. Application-specific middleware performs no domain-specific function by itself; instead, it provides abstract functional roles that can be played by other entities. These roles form adaptive structures, tailored to the particular application, for the composition and control of service interactions.

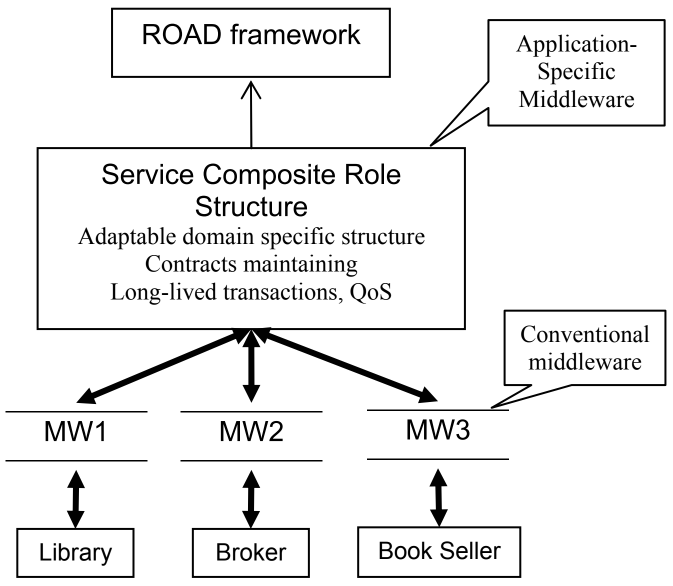

ROAD framework
Application-Specific Middleware
Service Composite Role Structure
Adaptable domain specific structure
Contracts maintaining
Long-lived transactions, QoS
Conventional middleware
MW1
MW2
MW3
Library
Broker
Book Seller

Figure 10-10: Schema service composite as application-specific middleware

As shown in Figure 10-10 above, application-specific middleware can be viewed as an extra layer that provides a level of indirection and management between services. This middleware consists of ROAD composite(s) of dynamically contracted roles which are played by the various services in the composition. The services that play roles can use whatever middleware standards they are built to, provided the service composite has adaptors that support those standards. The composite therefore functions as an interoperability bridge (Emmerich, 2000).

As well as performing the function of an inter-operability bridge, ROAD composites as middleware have the advantage of being able to:

- be distributed. Composites are, themselves, services that can be distributed.
- have each business organisation deploy and maintain its own composite that models its concerns. This would be the case in Figure 10-2 if the Libraries and the Broking Service had different owners. These middleware composites would communicate via their management interfaces, and via the functional role they play in each other's composite (the Library composite playing the Client role in the Broking Composite, and the Broking Composite playing the Supplier role in the Library composite). Ownership of composites is important because the ability to trigger a change in requirements or structure within a middleware composite raises the issue of which services have the authority to make changes in the composite. In our example, we would not want a book seller arbitrarily changing the terms-of-trade with a broker or client. The ability to distribute and ascribe ownership rights to composites helps organisations achieve secure cooperation in open environments.
- be recursively composed so that they can model complex multi-layered business domains
- have entities that map naturally to those domains. Composites can be used to model business entities, while contracts also map naturally to cross-organisational service-level agreements
- adapt to changing requirements and measured performance of the services that they compose.

To fully implement an application-specific middleware such as ROAD, further work needs to be done in integrating ROAD composites with standards for service discovery, coordination and the negotiation of service-level agreements. Research challenges also remain such as how to dynamically generate adaptors to overcome the heterogeneity of service middleware technologies; how to represent protocols of services (required order of exchanged messages); how to address different non-functional requirements such as security; and how to incorporate mechanisms for negotiation between composites that belong to different organisations.

10.8. Summary

The above case study presents a role-oriented approach to implementing a composite application that can maintain dynamic, yet long-term, relationships between commercial entities via Web service infrastructure. ROAD composites can be used to model role relationships between business functions, and these composites can be separately owned, deployed and maintained by different commercial entities. Abstract performative contracts that model business relationships suitable for creating virtual enterprises are defined. Because ROAD supports heterogeneous players, the model can include players that are Web services, software components and humans. Role models can therefore represent processes outside a computational domain; for example, a receiver player may do a physical check to see if the books ordered have been delivered. The case study also shows how non-numerical and multi-variant requirements can be passed as utility objects between composite organisers. ROAD composites can be viewed as a form of application-specific middleware that acts as an inter-operability bridge, that is fully distributable, and that maps well to business organisation and ownership domains.

11

Analysis and Discussion

This thesis presents both a conceptual meta-model for adaptive software systems (in Part II), and a framework for implementing that meta-model in a runtime system (Chapter 8). This chapter evaluates the ROAD framework both at the conceptual level as an approach to adaptive software architecture, and the framework's current implementation as a 'proof-of-concept' prototype.

The first section of this chapter evaluates how well the ROAD meta-model expresses those qualities necessary in an adaptive software system. The ROAD framework is evaluated in terms of the characteristics we used in Chapter 3 to discuss the various approaches to creating adaptive architectures.

The second section of the chapter discusses the prototype implementation in terms of the runtime overhead it imposes. ROAD defines an organisational middleware structure through which passes all communication between the application's functional runtime entities. This interposed message-intercepting structure creates an overhead compared to, say, the communication between two directly communicating objects. The run-time performance overhead of a ROAD application therefore needs to be characterised relative to such direct communication. We also compare the overhead imposed by ROAD middleware to the overhead imposed by Web service infrastructure, which is the predominant middleware for more open inter-organisational application integration.

11.1. Comparative expressiveness of the ROAD framework

In the literature review in Chapter 3, a number of adaptive software architectures were reviewed in terms of a range of criteria. In Section 3.3 these were categorised into criteria related to *(re)configuration* of the structure of the software, *regulation* of performance of non-functional requirements across that structure, and the nature of *management* of adaptation. In this section we will use these criteria (the italicised numbered points below) to characterise the ROAD framework, and to evaluate how expressive (capable) it is relative to other adaptive architectures.

1 Configuration

1.1. Reconfiguration possible at runtime.

A ROAD application can create (and destroy) two distinct types of connection at runtime: contracts between roles, and role-player bindings. It can also create new roles. These configuration mechanisms are defined in the composite's organiser role, and the decisions about reconfiguration are made by the organiser player. However, the current implementation of ROAD has a limitation in this regard, in that it relies on type-compatibility to ensure that structures are functionally well composed, rather than representing the composition using some underlying formalism (e.g. π calculus) that allows machine reasoning across the entire structure. It follows that the creation of new *types* of composition is a human design-time activity rather than an automated runtime activity, unless it can be supported by external mechanisms for ensuring functional compatibility such as WSDL/UDDI (not to mention semantic or behavioural compatibility) or unless compatibility can be checked by very smart organiser players.

1.2. Composition based on declarative description possible at runtime.

In its current implementation, ROAD roles, contracts and composites are Java types that need to be statically defined. However, these entities can be empty structures, the contents of which are dynamically created (e.g. terms are added to a contract at runtime, and roles and contracts are dynamically created by the composite). Instantiated ROAD composites can therefore be declaratively defined, subject to the types that are used in the structure being predefined. A future extension to ROAD could use the reflective capabilities of Java to declaratively define contract, role and composite types.

1.3. Functionally recursive structure.

One of the strengths of ROAD is that it allows recursive composition/decomposition of composites *at runtime*. Many dynamic architectures are only concerned with the dynamic *reconfiguration* of black-box components. In ROAD, on the other hand, composites are themselves players that play roles, and these composites can be replaced or modified at runtime. This facilitates dynamic *decomposition* as well as dynamic reconfiguration. In this sense, it is conceivable that, subject to sufficient intelligence in the organisers, an entire ROAD application could be redesigned at runtime, rather than being constrained by fixed role or component decompositions. ROAD adaptive composites can be created representing various levels of the decomposition at different levels of granularity, with each composite managed independently.

1.4. Non-functional restructuring supported.

In ROAD, roles are *instances*. It follows that multiple roles can be of the same type, be created in parallel, and yet serve a single functional purpose. The reality of varying role-player performance can therefore be reflected in the runtime configuration, rather than having roles represent only abstract function.

1.5. Elements can be substituted (indirection of instantiation supported).

Like most dynamic architectures, ROAD supports the runtime substitution of players/components. ROAD supports queuing to maintain *communication* state during structural reconfiguration and to cope with player absence (see point 1.8 below). However, because in ROAD it is the players who maintain *domain* state, further work needs to be done to ensure that players can be *safely* substituted, so that the domain state of the system as a whole is preserved.

1.6. Supports heterogeneous components.

ROAD is designed to facilitate the composition of heterogeneous players (components, services, agents, UIs). The adaptors that convert between different technologies are kept separate from the roles, and can be added dynamically to the composite. The ROAD prototype implementation currently can work with Java and Web service players. Further work needs to be done to extend the range of adaptors (e.g. CORBA, J2EE, FIPA-ACL), and to create mechanisms that can automatically generate adaptors that are type-compatible.

1.7. Structure is entirely defined and controlled by management.

ROAD maintains a strict separation between process (as performed by the players) and the structure of the roles and the contracts (as managed by the composite organiser). The fact that ROAD players are ignorant of the structure facilitates substitutability of those players. If a player needs to deal with multiple role-players of the same type, this needs to be done by using *data* based mechanisms such as tracking transaction IDs, rather than by players maintaining *structural* references to each other (e.g. in the previous chapter's Book Broker example, the agent playing the Shopper role would use

Order IDs to distinguish transactions and the *role* would be responsible for routing messages).

1.8. Partial instantiation possible.

Message queuing in ROAD roles allows an application that is built using the ROAD framework to continue to function at some level, even if not all players are present at any one time in the structure. This approach facilitates player substitution and could be used to support the lazy instantiation of players.

1.9. Formal reasoning about structure possible.

As mentioned in point 1.1 above, ROAD does not currently support formal reasoning across a composite structure, or the checking of compatibility other than basic type checking. In open systems, functional interfaces not only have to be syntactically compatible, but also need to be semantically and behaviourally compatible. Ensuring compatibility of such interfaces is still very much an open area of research. Preliminary work has been done on developing a relational structure of ROAD programming entities which will allow some checking for integrity (see the discussion of ROADmaker in Chapter 8.7), but this work is outside the scope of this thesis.

2 Regulation

2.1. Non-functional regulation possible.

Contracts are first class entities in ROAD, and provide the basis for implementing non-functional regulation. Contracts store non-functional requirements in the form of utility functions attached to contract terms. They also provide mechanisms that enable the measurement of performance with respect to those requirements. These requirements can be altered dynamically and transmitted between composites.

2.2. Control dynamics supported.

While control dynamics is not explicitly addressed in ROAD, the ability to pervasively define measurement points (and thus control variables) throughout the structure provides the basis for implementing control-theoretic concepts such as control of hysteresis and feedback. Domain-specific organiser *players* would function as dynamic controllers. The specification of the capabilities of these players is beyond the scope of the ROAD framework and this thesis.

2.3. Utility can be defined arbitrarily

The ROAD framework provides some generic time-based utility functions, but any utility function that the application programmer wants to define can be attached to the performance measurement points in a contract term.

2.4. *Utility requirements can be changed dynamically.*

In ROAD, the requirements of a contract can be dynamically changed by altering the utility function settings associated with a contract term. These utility functions are Java objects, and their parameters can be changed at runtime. These changes to NFRs are instigated by the composite organiser, who creates them in response to the NFRs it receives over its management interface.

2.5. *Type of utility can be changed dynamically.*

New types of ROAD utility function can be dynamically attached to contract terms at runtime. During an interaction, when a measurement point in a contract term is reached, any utility function attached to that point will be evaluated.

2.6. *Multi-dimensional utility supported.*

Multiple utility functions can be evaluated for contract transactions, and these functions can be aggregated, as discussed in the previous chapter. Organiser players could also be implemented that combine the performance measures from a number of contracts in their composite.

3 Management

3.1. *Mechanisms for determining the need for reconfiguration or regulation are defined.*

In ROAD, the mechanisms for monitoring performance are defined in contracts whose terms assess the utility of transactions with respect to contract requirements. This performance information is then passed to the organiser, whose player then applies strategies to determine whether or not regulation or reconfiguration is needed, based on the composite's obligations as defined in the role the composite plays in the 'enclosing' composite.

3.2. *Management as separate entity.*

Rather than being encapsulated in a single entity, ROAD management functions are performed by a separate system consisting of contracts, organiser roles, organiser players, and the management interface connections between composite organisers.

3.3. *Management exogenous versus endogenous.*

The ROAD management system can be exogenously imposed on the functional players without needing access to the internal implementation of those components. Measured performance in ROAD is always the measurement of a component's interactions *with respect to* the organisation, rather than the measurement of some internal assessment of performance. Required performance is defined in the terms of the contract(s) that obligate a role that is played by a player. However, while ROAD does not define how domain-specific players are implemented, some players may need to be sensitive to changes in their role's obligations.

3.4. Management distributed versus centralised.

ROAD management is distributed to the composite level with each composite having its own organiser. Organisers are only concerned with relations in their own composite, and are not aware of how other composites (e.g. sub-composites) are managed.

3.5. Management structure not subject to a single point of failure.

ROAD applications can be subject to a single point of failure to the extent that composites are constructed hierarchically. For example, if a composite player fails then all the sub-composites and players composed by the failed composite will be inaccessible to the enclosing composite. ROAD does not have (indeed very few other software architectures have) a distributed architectural description as found in (Georgiadis, 2002). However, ROAD composites could be built to cater for redundancy (for example having multiple players available to play the same role), and the management and reconfiguration capabilities of ROAD provide a natural way to implement reliable systems.

3.6. Separate management structure.

ROAD has a management structure in the form of a network of organisers. NFRs and performance data are both encapsulated in utility objects that flow over this network.

3.7. Management can find and/or select components (resolves indirection of instantiation).

In ROAD, organiser *players* are responsible for finding suitable candidate components, and selecting the best candidate. The specification of these domain-specific players is outside the scope of the ROAD framework.

3.8. Management mechanisms can be superimposed a posterior on functional components.

ROAD organisational structures can be superimposed on components that have not been designed to participate in such a structure. This presupposes that the requirements of roles in the structure are compatible with the pre-existing players, or that suitable adaptors are implemented to make the role compatible with the player.

3.9. Management updatable.

In ROAD, the management decision making process is the responsibility of organiser players. The ROAD framework does not define domain-specific organiser players or management strategies (sequences of reconfiguration operations in response to performance data and NFRs). In its current implementation, ROAD does not define any formalisms for representing strategies or methods for accessing such strategies within an organiser player. Improvement of strategies by learning could also be a possible attribute of organiser players. While such capability can be accommodated in the ROAD schema, it is outside the scope of the ROAD framework.

3.10. Management substitutable.

An organiser player can always be substituted with a more capable player, as the reflective representation of the composite and mechanisms for manipulating the structure are defined in the organiser *role*.

3.11. Supervisory control possible.

Supervisory control is a special case of organiser player substitution, as defined in the previous point. In this way management control can be overridden by external control (e.g. a human controller) if the circumstances warrant this (e.g. the automated organiser cannot find a configuration that meets the obligations of the composite).

3.12. Costs of reconfiguration estimated.

No in-built mechanisms for estimating or measuring the cost of reconfiguration are currently implemented in ROAD.

4 Other

4.1. Implementation apparent.

A proof-of-concept prototype of the ROAD framework has been implemented. Further work to be done on the framework is discussed in the final chapter.

Table 3.1 from Chapter 3 is reproduced below. This table compares adaptive architecture according to the above criteria and summarises the extent to which the ROAD framework meets the criteria.

Table 11-1: Summary of the characteristics of adaptive software frameworks

	Georgiadis	Plastik	Rainbow & SMs (Garlan,Cheng)	Rainbow (Huang)	Aura	Viable System Architecture	ConFract	CASA	ROAD
1. Configuration									
1.1. Reconfiguration possible at runtime.	✓	x	✓	~	✓	x	✓	~	✓
1.2. Declarative composition at runtime.	✓	x	x	x	x	x	✓	~	~
1.3. Functionally recursive structure	x	x	~	~	~	✓	✓	x	✓
1.4. Non-functional restructuring supported	~	x	~	~	~	x	✓	✓	✓
1.5. Elements can be substituted	✓	✓	✓	✓	✓	~	✓	✓	✓
1.6. Supports heterogeneous components	x	~	✓	✓	✓	x	x	✓	✓
1.7. Blind communication	x	x	x	x	x	x	x	x	✓
1.8. Partial instantiation possible	✓	✓	x	x	x	x	✓	x	✓
1.9. Formal composition	✓	✓	~	~	~	x	✓	x	x
2. Regulation									
2.1. Non-functional regulation possible.	x	~	✓	✓	✓	✓	✓	✓	✓
2.2. Control dynamics supported.	x	x	x	x	~	✓	x	x	~
2.3. Utility can be defined arbitrarily	x	~	✓	✓	✓	~	✓	x	✓
2.4. Utility requirements changed dynamically	x	x	x	~	✓	~	x	✓	✓
2.5. Type of utility changed dynamically.	x	~	x	~	~	~	~	✓	✓
2.6. Multi-dimensional utility supported.	x	~	✓	✓	✓	~	✓	✓	✓
3. Management									
3.1. Can determine the need for reconfiguration	x	~	~	~	✓	~	✓	✓	✓
3.2. Management as separate entity.	x	✓	✓	✓	✓	~	x	✓	✓
3.3. Management exogenous	x	~	✓	✓	✓	x	x	x	✓
3.4. Management distributed	✓	~	x	~	x	✓	✓	✓	✓
3.5. Management structure *not* subject to single point failure	✓	x	x		x	x	x	~	x
3.6. Separate management structure.	~	✓	~	✓	✓	✓	✓	✓	✓
3.7. Management can find /select entities	x	~	✓	~	✓	~	~	~	~
3.8. Management mechanisms superimposed	~	x	x	x	x	x	x	x	✓
3.9. Management is updatable.	x	✓	~	~	~	x	x	x	~
3.10. Management is substitutable.	x	✓	~	~	~	x	x	x	✓
3.11. Supervisory control possible.	x	x	~	x	✓	✓	~	✓	✓
3.12. Costs of reconfiguration estimable.	x	x	x	x	✓	~	x	x	x
4. Other									
4.1. Implementation is apparent.	✓	~	✓	✓	✓	✓	✓	✓	✓

Of the approaches to adaptive architecture discussed in Chapter 3, the most similar to ROAD are the Rainbow family (Garlan, Cheng, Huang et al., 2004) and ConFract (Collet, Rousseau, Coupaye et al., 2005). We will briefly compare these frameworks with ROAD.

Rainbow provides a reusable infrastructure with a management layer that models the application's architecture. This architecture layer has an *adaptation engine* and an *adaptation executor* that are similar, respectively, to ROAD's *organiser player* and *organiser role*. This adaptation engine makes decisions (selects strategies) on what needs to be changed on the basis of changes in the system layer indicated by monitoring probes. Like ROAD this management is exogenous, but in Rainbow the instrumentation is assumed to be embedded in the functional components, rather than being well-defined measurement mechanisms provided for in the infrastructure. In ROAD monitoring is integral to the organisational infrastructure in the form of contract terms. Management strategies in Rainbow are globally defined across the structure (similar to an architectural style). While it is conceivable that Rainbow applications could be modularised into self-managed composites like ROAD, there is no description of how such Rainbow modules could be composed, or how they would form an integrated adaptive system; as is enabled by the recursive composition of ROAD composites and the management system between composite organisers.

Like ROAD, ConFract is a contract-based framework that supports recursive composition. ConFract components have many similarities to ROAD composites. A ConFract component membrane is similar to the composite's external role. Both ConFract components and ROAD composites can be partially instantiated, and both have controllers. As described in Chapter 3, ConFract has a number of different types of component-wide contract whereas all contracts in ROAD are between two role instances. Like ROAD these contracts define both the structure and quality of interactions but, being component-wide, ConFract contracts, in the form of executable assertions, can express and enforce more complex behavioural dependencies between subcomponents. In ROAD these assertions (in the form of contract terms) are distributed down into the binary contracts by the composite organiser. In ConFract, specifications (and thus contracts) can be related to the external interface of a component, or the internal and external parts of a composite. ROAD, on the other hand, radically separates external function (role) from implementation (player). In ConFract, replacing an implementation (player) involves generating a new contract(s) from a new or existing specification. While it is feasible, indeed desirable, in ROAD to implement the declarative definition of self-managed composites, this is not yet supported in the current implementation of the ROAD framework. ConFract emphasises the definition of constraints in contracts at design-time, however details of its runtime management mechanisms have not, as yet, been published in the English language journals.

11.2. Runtime overhead

In this section we examine the ROAD framework prototype in terms of the runtime overhead it imposes. ROAD defines an organisational middleware structure through which passes all communication between the application's functional runtime entities. This message-intercepting interposed structure creates some overhead.

11.2.1. Factors in overhead

To provide a preliminary characterisation of the overhead imposed by the ROAD framework, we compare a method invocation through the ROAD framework with a direct method invocation from one object to another. In general there can be a number of sources of overhead that arise from the ROAD framework. These extra steps can include:

1. Passing the invocation via a role
2. Interception method invocation
3. Invoking *before* and *after* aspect advice to call any utility functions
4. Calculating utility functions associated with an interaction
5. Marshalling of methods into messages and queuing them in the role
6. Converting messages from Java into external formats e.g. SOAP.

There are also a number of overheads involved with reconfiguration operations, such as the time it takes to create new contracts, bind players to roles, discover and evaluate potential players and so on. Such restructuring operations are likely to be 'out-of-band' (Coulson, Blair and Grace, 2004); that is, relatively infrequent compared to operational interactions. Although they may be relatively infrequent, these reconfiguration overheads will still need to be characterised in a domain-specific application, as the organiser players will need to know these costs to enable them to perform cost-benefit analyses of adaptation strategies. However, we will not attempt to characterise the performance of these reconfiguration operations here because the costs of creating the internal configuration in the ROAD composite (e.g. adding a role, creating a new contract, referencing a new player) are likely to be insignificant compared with the costs of creating the external links to domain specific players (e.g. service discovery, SLA negotiation, adaptor generation, and so on.). The functions and the strategies to implement reconfiguration are the responsibility of organiser *players* and, as such, are outside the scope of this thesis.

11.2.2. ROAD tests

To quantify the relative significance of different sources of overhead in the ROAD framework, a number of tests were run. These tests entailed invoking a standard asynchronous transaction (an invocation of a method and response). In each type of test this transaction passed through different, and increasingly more complex, configurations of the ROAD middleware framework. For each configuration the standard method was invoked in a loop multiple times (100 to 100,000 iterations depending on the speed of processing the invocation) and the total time to process all iterations was measured. This total time was divided by the number of iterations in order to calculate the duration of a single transaction. This process was repeated multiple times to obtain a mean measurement for the invocation over each of the middleware configurations. The standard method invoked was kept very small (a simple arithmetic calculation performed 100 times), so that any change in overhead imposed by the middleware would be relatively large, and thus could be detected. All tests were performed on the same machine[1], and Web service invocations were executed on a server[2] running as a local host to remove any routing or network variability.

All tests invoked the same method but the test cases progressively added ROAD features as listed in the previous subsection. The test configurations are listed below with an identifying name and a short description.

1. **01_Object-to-Object**. This is a benchmark test consisting of a standard invocation of a method from one object to another, and its asynchronous response.
2. **02_RolePlayer-RolePlayer.** In ROAD, all invocations pass through roles, which involves an extra method call.
3. **03_Contract**. In this test all method invocations between roles pass through a contract which intercepts the message and checks that it is authorised under the terms of the contract (using pointcut matching defined in the aspect).
4. **04_ContractWithAdvice**. In this test a simple time-based utility function is invoked in the before and after advice of each transaction to measure how long the transaction takes. Updating the utility function involves the overhead of reflectively accessing the joinpoint's execution context, and passing details of the method signature and the invoking object to the utility function object that is attached to the contract term.

[1] Pentium 4, 3.0 Ghz, Hyper Threading 512 Mb RAM running Windows XP SP2, Java 2 Runtime Environment, SE, HotSpot™ Client VM (1.5.0_06-b05)

[2] Server running Tomcat v 5.5.16, Apache Axis2, SOAP v1.2, data binding XMLBeans, client code generated by Axis2 WSDL2Java tool.

5. **05_MessageQueue**. In this test the receiving role encapsulates the request invocation into a message object, and stores it in a queue. A message processor in the role passes the message to its role player when it is ready.
6. **06_WS_NoQueue.** In this configuration the target method resides on a Web server. The target role delegates the invocation to a Web service proxy that uses a Java to SOAP engine to convert and send the message to the Web server. The Web server then de-marshals the SOAP message. This process then operates in reverse for the response message.
7. **07_WS_Queue.** This is the same as the previous case except that the target role buffers incoming messages in a queue before passing them to the Web service proxy player.

11.2.3. Results and discussion

Each run of a test involved the repeated invocation of a transaction (request and response) over a configuration of ROAD middleware, as shown in Table 11-2 below. The total time to execute the test run was divided by the number of transactions in order to calculate the duration of a single transaction. Each run of transactions was repeated 50 times, and the mean duration of each transaction was calculated. The table also includes a column that indicates the relative overhead cost of each of the configurations. These ratios are based on the invocation of a very small process in the player, and will, of course, decrease as the time taken to execute the process increases relative to the overhead. Variation between runs of the same test configuration, as measured by the coefficient of variation ($\sigma/\mu*100$), was less than 10% in non-Web service transactions, increasing to less than 20% in Web service transactions.

Table 11-2: Mean time in milliseconds and relative duration to perform a standard transaction for each middleware configuration

Configuration	Transactions per test run	Mean Time (msec) per transaction	Time relative to test 1	Coefficient of variation
01_Object-to-Object	100,000	0.00094	1	5.3%
02_RolePlayer-to-RolePlayer	100,000	0.00094	1	6.9%
03_Contract	10,000	0.01316	14	2.0%
04_ContractWithAdvice	10,000	0.06110	65	7.7%
05_MessageQueue	10,000	0.11355	121	9.9%
06_WS No Queue	100	7.70700	6632	17.3%
07_WS Queue	100	8.58455	7017	19.5%

As can be seen in Table 11-2, passing invocations via roles (Test 2) imposed no detectable extra overhead relative to the Object-to-Object reference test (Test 1). Standard ROAD features implemented in Test 3, 4 and 5 (interception contracts, performance measurement and queuing, respectively) imposed successively greater overheads, up to approximately 1/10 millisecond per asynchronous transaction.

Test 3 is representative of the case where contracts are used to ensure that only authorised transactions occur between roles. These results indicate the relative efficiency of using association aspects pointcuts to implement method interception (as opposed to message interception and inspection), an efficiency which is not surprising given the interception code is woven into the role's code at compile time. Association-aspects do impose a small additional overhead viz ordinary AspectJ aspects, as the association-aspect is a separate object, but Sakurai (Sakurai, Masuhara, Ubayashi et al., 2006) has shown that this overhead is relatively small.

Test 4 is representative of the case where contracts are used to measure the performance time of a contracted interaction. The increase in overhead (~ 5 times the non-measurement case) is due to the need to access the execution context. This context is accessed in order to obtain a reference to the contract parties, to obtain a measurement of the system time, and to update the utility function. If the domain application requires the calculation of more complex utility functions with each interception, then an additional overhead would be imposed.

Use of queuing (Test 5) approximately doubles the overhead relative to Test 4. In domains where high speed performance is a requirement, consideration might be given to the judicious use of message queuing. For example, queues might only be used in those roles that are required to store communication state, or an organiser might only activate a queue when it is about to swap a player.

However, as can be seen from Figure 11-1 below, the overhead imposed by the ROAD middleware as indicated by Tests 3, 4 and 5, is insignificant relative to the cost imposed by the Web service infrastructure, even when network transmission costs are removed. In a Web service context, ROAD middleware only accounts for between 0.9% (without queue) and 1.7% (with queue) of the total middleware overhead.

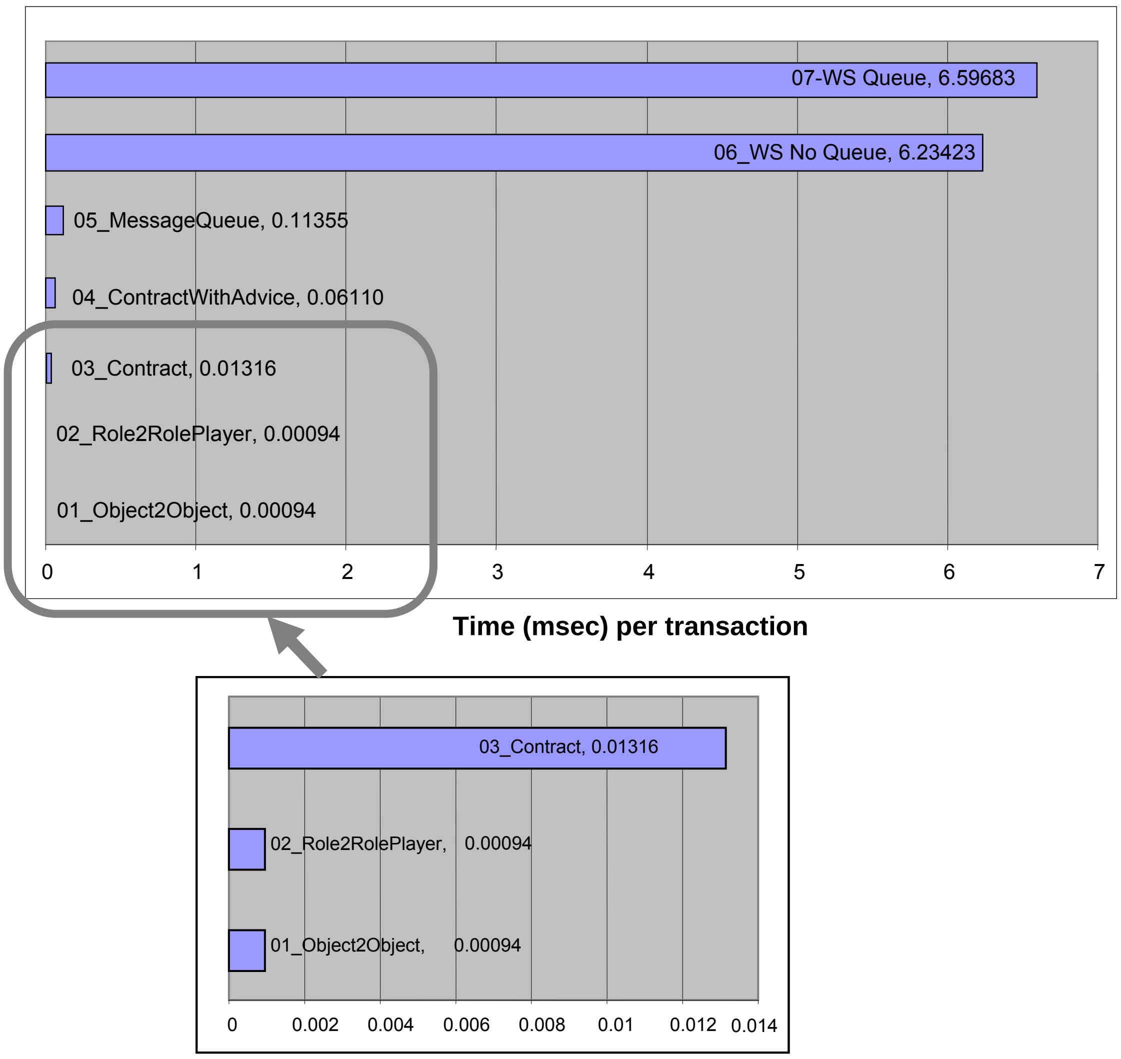


Figure 11-1: Mean duration per transaction (msec) for different configurations of ROAD and Web Service middleware

We can conclude that using ROAD in a Web service context imposes no significant overhead relative to the Web service middleware. As the maximum absolute overhead (ignoring the cost of any domain-specific utility functions) was of the order of a 1/10 of a millisecond per transaction on a standard PC, ROAD could also be effectively used in situations where the processing time of the players is significantly greater than this value (e.g. manufacturing control).

11.3. Summary

In terms of the characteristics of adaptive architectures described in Chapter 3, the properties of an application built using the ROAD framework can be summarised as follows. The ROAD framework supports the heterogeneous composition of self-managed composites. ROAD composites support two type of reconfiguration, namely limited non-functional reconfiguration of the role-structure and functional component (player) substitution. A composite role-structure does not always have to have all of its roles assigned to players, in order for the composite to be viable. ROAD also supports regulation of composites through the ability to dynamically set the terms of contracts that connect roles, and by providing in-built monitoring mechanisms that can check if the actual performance of role-players meets the requirements defined in their contracts. The utility functions used to monitor performance can be dynamically changed at runtime. The management in ROAD is exogenous in that management structures can be superimposed onto pre-existing components, and no access is required to the internals of those components. ROAD composites can be recursively composed with each composite in the composition at a different level of granularity. The self-managed nature of ROAD composites can provide a basis for reliable systems that can handle complexity by limiting the concerns of any one organiser to a role composition at a single level of abstraction. The overall regulatory behaviour of an application is achieved by requirements and performance data flowing over a management system between composite organisers. Organiser players are separable from the roles they play, allowing players to be substituted or upgraded to more capable players with different adaptive strategies.

At runtime, the ROAD middleware does impose a performance penalty because the interactions between components are intercepted and may be monitored for performance. Message queuing mechanisms also impose an overhead. Depending on the configuration, this total overhead is up to approximately 120 times a standard asynchronous transaction between two Java objects. In absolute terms, this overhead was approximately 1/10 millisecond per asynchronous transaction on the testbed PC. However, in the context of Web services technologies which are becoming the industry standard for service composition, the overhead of the ROAD middleware framework is relatively insignificant, being less than 2 percent of the total overhead incurred by Web service middleware.

12

Conclusion

This chapter concludes the thesis by discussing its contribution to research into adaptive software systems. We then discuss the future work that could to be done to further develop the ROAD approach to developing ontogenically adaptive software.

12.1. Contribution

The major contribution of this thesis is to show how adaptive software systems can be devised that can respond to both changes in requirements and to changes in the environments in which these systems operate. A novel framework has been developed to facilitate the creation of adaptive and adaptable applications that are scalable, distributed, grounded, recursively structured and self-managed. A summary of the key characteristics of this ROAD framework, and how it compares with other adaptive architectures, can be found in section 11.1 of the previous chapter. We have demonstrated how an application built on this framework can adaptively restructure itself in response to changing conditions.

As well as developing the ROAD framework, a number of additional contributions have been made to software engineering both at conceptual and technical levels. These are listed below.

This thesis shows how the biologically-based concepts of ontogenic adaptation and organisation can be applied to software. The concept of *adaptation* is often used in a confused or muddy way in software engineering. Different types of adaptation need to be clearly distinguished. This thesis draws on the work on biological cognition by Maturana and Varela (1980) to distinguish ontogenic adaptation from other types of adaptation (evolutionary adaptation and environmental manipulation), and shows how the concept of ontogenic adaptation can be applied to goal-directed software systems. Key to this definition of ontogenic adaptation is the concept of *autopoiesis*, i.e. the maintenance of organisational relationships within a viable system. This thesis proposes that software systems should be viewed as *organisations* that have as a primary goal the maintenance of such homeostatic relationships both within the system, and between the system and its environment. We introduce an analytical framework for evaluating to what extent adaptive software architectures conform with the principles of ontogenic adaptation.

The conception of organisation in this thesis is based on three principles. The first two principles derive from systems theory: the first being the separation of control from process, and the second the being the recursive distribution of control down through the organisational structure (or viewed from the bottom up, the creation of successively higher levels of control). This thesis proposes a third principle: the strict separation of a *role* from the *player* that executes that role. Software organisations are viewed as dynamic loosely-coupled role structures analogous to a human organisation. While other approaches (e.g. (Herring, 2002; Cai, Cangussu, DeCarlo et al., 2004)) have proposed a cybernetic view of software as a control system, to the best of our knowledge the role-based organisational view propounded here is unique.

This thesis clarifies the concept of software roles by distinguishing between (i) roles as descriptors for association-ends, (ii) player-centric roles that add functionality to a stable entity, and (iii) organisation-centric roles that represent stable abstract functions (goal-oriented positions) within an organisational structure. These are distinct concepts that are often blurred or confused in the literature on software roles. While the concept of an organisation-centric role is not unique to this thesis, in that it appears in some object-oriented and agent-oriented literature (e.g. (Baldoni, Boella and van der Torre, 2005b; Odell, Nodine and Levy, 2005; Zambonelli, Jennings and Wooldridge, 2000)), the conception of functional roles in this thesis is distinct from other approaches. ROAD role instances are middleware runtime entities that define an abstract function, perform messaging and queuing functions, but do not, in themselves, execute any domain function. While other role-based views of software organisations have been developed (e.g. (Baldoni, Boella and van der Torre, 2005a; Herrmann, 2005)) these approaches are not *per se* adaptive.

Complex human organisations typically define roles that allow the players varying degrees of autonomy and thus require varying capability from their players. Likewise, players in different roles within a software organisation may be heterogeneous (objects, components, services, agents or humans), and have very different degrees of autonomy and capability. Indeed, we argue that this differentiated autonomy is a necessary attribute of complex, goal-directed organisations (Colman and Han, 2005). This thesis proposes a novel conceptual scheme for defining the relationship between roles and players, based on five levels of autonomy that roles can allow their players. We show how these levels effect the implementation of roles and players. As software systems become more complex and open, composite structures will need to facilitate such heterogeneity.

Roles in ROAD are associated by *contracts*. This thesis introduces a novel conception of software contracts. The concept of a contract has often been used in software engineering, predominately as means to enforce conditions at the interface of a component (e.g. (Meyer, 1988)). As discussed in Chapter 3, contracts have also been used as a means of composition in adaptive architectures (e.g. (Collet, Rousseau, Coupaye et al., 2005)). A service level agreement that sets out QoS requirements can also be thought as a type of inter-organisational contract, such as can be specified in WSLA (IBM Corporation, 2003). ROAD contracts are *instances* of binary associations between role instances that combine aspects of all of the above conceptions; namely, interaction control, composition, and performance. ROAD contracts exist as a *type*, as an individualised *specification*, and as an implementation *entity* that monitors and can enforce aspects of the specification. This thesis also introduces the concept of contract abstraction. Abstract performative contracts allow common interaction patterns to be defined and reused. The performative contracts provided by the ROAD framework can also be extended by the application programmer to create domain-specific contracts that can then be instantiated with concrete contracts.

At the technological level, this thesis shows how *association aspects* (Sakurai, Masuhara, Ubayashi et al., 2006), can be used to implement ROAD contracts. While aspects have previously been suggested as a means of implementing interface-enforcement contracts (e.g. (Kendall, 1999)), in order to implement ROAD contracts we need to be able to create aspect *instances* that associate groups of (role) objects. Association-aspects provide this facility, and, as such, this thesis provides a novel use of association aspects.

This thesis also proposes a novel management system constituted of connected organisers. Non-functional requirements and performance data flow over this network of organisers. A ROAD management system is analogous to a business-management system in that it is separate from the functional system (i.e. the processes executed by the functional role-players), yet this management system can control individual relationships in the functional system (i.e. by controlling the contracts between roles).

The connecting points between the management system and functional system (process) are the contracts that the organisers control. While many adaptive software architectures (as described in Chapter 3) propose a separate management layer, such layers are typically implemented through the global application of policies that are targeted at the components rather than the connections. In ROAD, management is distributed down to the organisers of composites, and these organisers create and regulate the contracts. The type of management system used in ROAD enables the creation of recursive and distributed organisations that scale well and decompose complexity into manageable modules connected by a structure (Simon, 1969).

This thesis also shows how monitoring can be done via the contracted role relationships. This facilitates the creation of management systems that are exogenous to the functional components, and can be imposed on these components *a posterior*. Monitoring via relationships gives ROAD the added advantage of monitoring mechanisms provided by the framework, rather than relying on monitoring mechanisms being built into the components as in other adaptive architectures (e.g. (Garlan, Cheng, Huang et al., 2004)).

Finally, in the context of service-oriented computing, this thesis introduces the concept of an adaptive *application-specific* middleware. Such middleware acts as an interoperability bridge that can model and implement long-term inter-organisational relationships between heterogeneous services. When used with Web service technologies, we have shown that the overhead imposed by this ROAD-based adaptive middleware is insignificant, relative to the overheads imposed by Web service middleware.

12.2. Future work

The ROAD meta-model presents a broad vision of what it means for a software application to be ontogenically adaptive. The ROAD framework, as it is currently implemented, supports the key concepts of the ROAD meta-model. However, there remains much work to be done to enhance the framework and to develop tool support, before such an approach can become a widely accepted engineering reality. Throughout the body of this thesis, we have pointed to areas of further research and development that could be pursued. In particular, Section 8.6 discussed a number of aspects of the ROAD framework that need to be further developed in order to fully realise ontogenic adaptation in software applications. To summarise, this work includes:

- Mechanisms for the aggregation of NFRs found in contracts into role "position descriptions".

- Support for protocol and general clauses in ROAD contracts including the development mechanisms for the tracking protocols consisting of a number of transactions. Because ROAD contracts are binary, ways of coordinating interactions involving more than two parties need to be defined at the composite-organiser level.
- Development of additional organisational 'architectural styles' that can be expressed in terms of arrangements of abstract performative contracts. In our thesis the Widget and Broker organisation examples can be viewed, respectively, as command-hierarchy and supply-chain structures. Other types of abstract organisational structure could also be expressed, for example, pull-driven organisations that are controlled through resource constraints.
- Integration with other middleware technologies and standards, in particular the development of role-player adaptors that can act as inter-operability bridges with technologies such as CORBA or agent communication languages.
- Further development of the association-aspect mechanisms that underpin ROAD contracts, to support dynamic deployment of new types of contracts, and the ability to denote pointcuts on method annotations so that that role method names do not have to be mangled in order to associate them with CCA abstract message types.
- Development of a generic language for the communication of NFRs and performance measures between organisers (only the form of the messages as utility objects has thus far been defined).
- Mechanisms to insure player transition can occur safely and without loss of domain state.
- Support for the declarative definition of organisational structures, and the deployment of declaratively defined organisations.

In ROAD, the decisions about when and how to regulate or reconfigure a composite are the responsibility of the composite's organiser player. The definition of these domain-specific players is outside the scope of the ROAD framework and this thesis. That being said, an organiser's capability is core to the adaptive capability of the composite. It is therefore appropriate to mention the major research challenges that need to be addressed in developing such players. These challenges include:

- Defining (a library of) strategies for composite adaptation (like that illustrated in Figure 7-4). These strategies would include control-theoretic strategies based on characterisations of the cost of reconfiguration operations.

- Defining formalisms and strategies for the valid and automatic decomposition of composite level NFRs into contracts terms between roles in that composite.
- Implementing capabilities in the organiser that will allow composites to act as players in open agent or service-oriented contexts. Depending on the application domain, these capabilities might include player (agent or service) discovery. This would require organiser players be able to use standardised player discovery mechanisms (e.g. UDDI). Organisers may also need to perform service-level (QoS) and protocol negotiation, which would necessitate that internal ROAD contract representations be mapped to external standards for defining service-level agreements (e.g. WS-Agreement) and collaboration (e.g. WS-Coordination). In open or inter-organisational contexts security issues may also need to be addressed.
- The current implementation of ROAD, compositions are limited to predefined contract types. These compositions do not define new functions. If interfaces could be completely described at syntactical, behavioural, QoS and semantic levels, then sufficiently intelligent organisers may be able to perform automated functional composition.
- Advanced organisers need to be able to model the cost of reconfiguration to determine if the benefits of the new configuration outweigh the costs of achieving that new structure. For an organiser to do this, it needs to maintain dynamic models of the organisation. These dynamic models would be a prerequisite for developing advanced control-theoretic software.

As pointed out in Chapter 8 (Section 8.7), tools need to be developed to facilitate the development of ROAD organisational structures and to check the consistency of those structures. This will require the development of a reflective meta-model of ROAD programming constructs (roles, contracts, composites etc.) that will allow reasoning about the organisational structures that are defined using those constructs. Development methodologies appropriate to ROAD also need to be elaborated. It would be beneficial to integrate ROAD with existing SDLC (Software Development Life-Cycle) methodologies that provide role-oriented analysis techniques (e.g. (Reenskaug, 1996; Juan, Pearce and Sterling, 2002; Wooldridge and Jennings, 2000)).

Finally, the applicability of ROAD to various application domains needs to be further explored. In this thesis we have described the use of ROAD to model two types of application domain: a manufacturing control system, and a service-oriented supply chain. Other types of domain could prove amenable to ROAD. These include domains such as pervasive computing and mixed-initiative control-systems that are characterised by volatile requirements, components and environments.

We began this thesis by showing how principles of systems-theory can be applied to the design of adaptive software architectures. Being based on these principles, the ROAD framework provides a way of creating organisational structures and provides mechanisms for regulating interactions over those structures. These regulated organisational structures provide a higher level of abstraction at which software can be conceptualised, and open possibilities for new research directions. Such organisational abstractions of software systems might provide the basis for the development of a theory of software organisation, just as a large body of management theory has been developed focussed on human organisations. With such a management theory we may begin to move beyond the basic principles of software design, such as low coupling and high cohesion, and beyond simple architectural patterns and styles, towards a more complete understanding of what makes a good software organisation: one that is both goal-directed and viable in open and dynamic environments.

Appendix

Test Harness Code and Output

INPUT

OUTPUT

```
package widgetOrg;

import mContract.Composite;
import mContract.Organiser;
import widgets.Foreman;
import widgets.LazyEmployee;
import widgets.Manager;
import widgets.NonFnRole;
import widgets.ProductionManager;
import widgets.SkillfulEmployee;
import widgets.ThingyMaker;
import widgets.WidgetMaker;

/**
* A test program to test the functionality of a composite, and the abilities
* to swap players and to create new role at run time of organiser.
*
* @author Alan Colman
* @author Linh Duy Pham
*/
public class TestComposite
{
public static void main(String[] args)
{
    ProductionManager pm = new ProductionManager("Production Manager");
              WidgetMaker wm = new WidgetMaker("Widget Maker");

              //player for ProductionManager
              Manager manager = new Manager("Manager");

    // Organiser setup
              WidgetDepOrganiserPlayer orgPlayer = new WidgetDepOrganiserPlayer();
              Organiser org = new WidgetDepOrganiser(new WidgetDepRoleFactory());
```

INPUT **_OUTPUT_**

```
org.setPlayer(orgPlayer);

// Player for WidgetMaker --> create the WidgetDepComposite
Composite widgetDepComposite = new WidgetDepComposite();
org.setComposite(widgetDepComposite);

//create ThingyMaker and Foreman
ThingyMaker t = new ThingyMaker("Thingy Maker");
Foreman f = new Foreman("Foreman");

//create other players
SkillfulEmployee foremanPlayer = new SkillfulEmployee("Foreman/Thingy
            Player");
SkillfulEmployee goodThingyMakerPlayer = new SkillfulEmployee("Skillful
            ThingyMaker");
LazyEmployee badThingyMakerPlayer = new LazyEmployee("Lazy Thingy
            Maker");

System.out.println("\nThere are three ThingyMaker players created.");
System.out.println("Foreman/Thingy Player: performance 20 ms");
System.out.println("Skillful Thingy Maker: performance 20 ms");
System.out.println("Lazy Thingy Maker: performance at start is 10 ms, increase
            by 20 ms every time a thingy is made, with cap of 100
            ms");

// add Roles and Players to composite
widgetDepComposite.addRole(f);
widgetDepComposite.addRole(t);

widgetDepComposite.addPlayer(foremanPlayer);
widgetDepComposite.addPlayer(goodThingyMakerPlayer);
widgetDepComposite.addPlayer(badThingyMakerPlayer);
```

```
There are three ThingyMaker players created.
Foreman/Thingy Player: performance 20 ms
Skillful Thingy Maker: performance 20 ms
Lazy Thingy Maker: performance at start is 10 ms, increase by 20 ms every time a thingy is
            made, with cap of 100 ms

```

INPUT

```
        // Setup Initial Players
        try
        {
            //ProductionManager
            pm.setPlayer(manager);
            //WidgetMaker
            wm.setPlayer(widgetDepComposite);

            //Foreman and ThingyMaker
            f.setPlayer(foremanPlayer);

            System.out.println("\nStart up program with 1 ThingyMaker role, Lazy
                        Thingy Maker is the initial player.");
            t.setPlayer(badThingyMakerPlayer);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }


        System.out.println("\n\n----------------------------------------------------");
        System.out.println("---- Before contract between Production Manager and
                        WidgetMaker is created ----");
        System.out.println("TEST: Should have error non contracted between
                        ProductionManager and WidgetMaker");

    try
        {
            pm.do_placeOrderWidgets();
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
```

OUTPUT

```
Start up program with 1 ThingyMaker role, Lazy Thingy Maker is the initial player.



-----------------------------------------------------
---- Before contract between Production Manager and WidgetMaker is created ----
TEST: Should have error non contracted between ProductionManager and WidgetMaker
To user: Enter number of widgets required: 15
X--X CCA call from uncontracted functional role: call(void
                              widgets.WidgetMaker.do_makeWidget(int))

```

INPUT

```
e.printStackTrace();
}

System.out.println("\n\n--------------------------------------------------");
System.out.println("---- Creating contract between ProductionManager and WidgetMaker ----");
// create contract
ProManagerWidgetMakerContract contract = new ProManagerWidgetMakerContract(pm, wm);

System.out.println("\n\n----------------------------------------------------");
System.out.println("---- Before contract between Foreman and ThingyMaker is created ----");
System.out.println("TEST: Should have error non contracted between Foreman and ThingyMaker");

try
{
          pm.do_placeOrderWidgets();
}
catch (Exception e)
{
          System.out.println(e.getMessage());
          e.printStackTrace();
}

System.out.println("\n\n----------------------------------------------------");
System.out.println("---- Create contract between Foreman and ThingyMaker ----");
org.createContract(f, t);

System.out.println("\n\n----------------------------------------------------");
System.out.println("---- Now place an order of widget ----");
System.out.println("TEST: Should produce widgets");
```

OUTPUT

```

----------------------------------------------------
---- Creating contract between ProductionManager and WidgetMaker ----
do_makeWidget term added to contract
qry_widgetOrder term added to contract



----------------------------------------------------
---- Before contract between Foreman and ThingyMaker is created ----
TEST: Should have error non contracted between Foreman and ThingyMaker
To user: Enter number of widgets required: 15
---> before a1 do AtoB : call(void widgets.WidgetMaker.do_makeWidget(int)) – Calculate Start time.
---> after a0 error: call(void widgets.WidgetMaker.do_makeWidget(int))
X--X CCA call from uncontracted functional role: call(void widgets.ThingyMaker.do_makeThingy())


----------------------------------------------------
---- Create contract between Foreman and ThingyMaker ----
Utility: FTUtility: thingiesPerSec0
do_makeThingy term added to contract
inf_thingyMade term added to contract
----------------------------------------------------
 ---- Now place an order of widget ----
 TEST: Should produce widgets
```

INPUT

```

            try
            {
                pm.do_placeOrderWidgets();
            }
            catch (Exception e)
            {
                System.out.println(e.getMessage());
                e.printStackTrace();
            }
        }

    }
============= END OF INPUT ==========================================
```

The rest of the output demonstrates the adaptive behaviour of the composite as the performance of the players change.

OUTPUT

```
To user: Enter number of widgets required: 15
---> before a1 do AtoB : call(void widgets.WidgetMaker.do_makeWidget(int)) – Calculate
                              Start time.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                              time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                              time.
Thingy made by Lazy Employee named Lazy Thingy Maker, in 10 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 15 msec
Moving average is 15.0 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) - CalculateStart
                              time.
Thingy made by Lazy Employee named Lazy Thingy Maker, in 30 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
```

```
afterUpdate Last elapsedtime 32 msec
Moving average is 23.5 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                              time.
Thingy made by Lazy Employee named Lazy Thingy Maker, in 50 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 94 msec
Moving average is 47.0 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is in breach
IN BREACH

*** In Breach Action

*** Replaced by a better player
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                              time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 93 msec
Moving average is 93.0 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is in breach
IN BREACH

*** In Breach Action
Utility: FTUtility: thingiesPerSec0
do_makeThingy term added to contract
inf_thingyMade term added to contract

NOTE: New contract is created between object of class widgets.Foreman and object of class
                              widgets.ThingyMaker
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                              time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms
```

OUTPUT

```

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 63 msec
Moving average is 78.0 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                                        time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                                        time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 63 msec
Moving average is 73.0 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
```

OUTPUT

```
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 31 msec
Moving average is 31.0 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                                        time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                                        time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 62 msec
Moving average is 70.25 msec
calculateUtility
```

OUTPUT

```
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 31 msec
Moving average is 31.0 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 62 msec
```

OUTPUT

```
Moving average is 68.6 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 31 msec
Moving average is 31.0 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start time.
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

```

OUTPUT

```
<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 63 msec
Moving average is 67.66666666666667 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
To user: Thingies are made. Quantity = 1
Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 32 msec
Moving average is 31.25 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.

---> before a1 do AtoB : call(void widgets.ThingyMaker.do_makeThingy()) – Calculate Start
                                        time.
---> after a0 : call(void widgets.ThingyMaker.do_makeThingy())
signature: do_makeThingy
Message added.
---> after a0 : call(void widgets.WidgetMaker.do_makeWidget(int))
signature: do_makeWidget
Thingy maded by Skillful Employee named Skillful Thingy Maker, in 20 ms
```

OUTPUT

```

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 31 msec
Moving average is 31.2 msec
calculateUtility
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
Thingy maded by Skillful Employee named Foreman/Thingy Player, in 20 ms

<--- before b1 : call(void widgets.Foreman.inf_thingyMade())
afterUpdate Last elapsedtime 63 msec
Moving average is 67.0 msec
calculateUtility
widgets.ThingyMaker:Thingy Maker: do_makeThingy is underperforming
UNDER-PERFORMANCE
To user: Thingies are made. Quantity = 1
<--- after b0 : call(void widgets.Foreman.inf_thingyMade())
signature: inf_thingyMade
```

Author Index

References

Agha, G.A. (2002) Adaptive Middleware - Introduction. *Communications of the ACM* vol 45, no 6, pp 31-23.

Agre, P.E. (1995) Computational research on interaction and agency - Introduction. In: Agre, P. and Rosenschein, S.J., (Eds.) *Computational theories of interaction and agency*, pp 1-52. MIT Press

Aldrich, J., Chambers, C., and Notkin, D. (2002) ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering ICSE '02* Orlando, Florida, New York, NY: ACM Press.

Aldrich, J., Sazawal, V., Chambers, C., and Notkin, D. (2002) Architecture-centric programming for adaptive systems. In *Proceedings of the First Workshop on Self-Healing Systems WOSS '02* Charleston, South Carolina, New York, NY: ACM Press.

Allen, R. and Garlan, D. (1997) A formal basis for architectural connection. *ACM Transactions in Software. Engineering Methodol.* vol 6, no 3, pp 213-249.

Allen, R.J., Douence, R., and Garlan, D. (1998) Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE '98)*

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web services concepts, architectures and applications*, Berlin, London: Springer.

Andrade, L., Fiadeiro, J.L., Gouveia, J. and Koutsoukos, G. (2002) Separating computation, coordination and configuration. *Journal of Software Maintenance and Evolution: Research and Practice* vol 14, no 5, pp 353-369.

Apache Web Services Project (2006) Axis Web Sevices, *http://ws.apache.org/axis/,* Last accessed: Jul 2006

Arbab, F. (1998) What Do You Mean, Coordination? *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)* vol March,

Ashby, W.R. (1956) *An introduction to cybernetics*, London: Chapman & Hall.

Baldoni, M., Boella, G., and van der Torre, L. (2005a) Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *Proceedings of PROMAS workshop at AAMAS'05*

Baldoni, M., Boella, G., and van der Torre, L. (2005b) Roles as a coordination construct: Introducing powerJava. In *In Procs. of MTCoord'05 workshop at COORD'05,*

Baldoni, M., Boella, G., and van der Torre, L. (2005c) Introducing Ontologically Founded Roles in Object Oriented Programming: powerJava. In *AAAI Fall Symposium, Roles, an interdisciplinary perspective* Arlington, Virginia, AAAI Press.

Baresi, L., Ghezzi, C., and Guinea, S. (2004) Smart Monitors for Composed Services . In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)* New York, NY, USA , ACM Press.

Barros, A., Dumas, M. and ter Hofstede, A. (2005) Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. *Technical Report FIT-TR-2005-02 Faculty of Information Technology, QUT, Australia.*

Batista, T., Joolia, A., and Coulson, G. (2005) Managing Dynamic Reconfiguration in Component-based Systems. In *Proceedings of the European Workshop on Software Architectures* Pisa, Italy,

BEA Systems, IBM, and Microsoft (2004) Web Services Coordination (WS-Coordination) http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf.

BEA Systems, IBM, Microsoft, SAP, AG, and Siebel Systems (2003) Business Process

Execution Language for Web Services (BEPL4WS) http://ifr.sap.com/bpel4ws/index.html.

Beer, S. (1979) *The heart of enterprise*, Chichester Eng., New York: Wiley.

Beer, S. (1984) The Viable System Model: Its Provenance, Development, Methodology and Pathology. *Journal of the Operational Research Society* vol 35, no 1, pp 7-25.

Beer, S. (1985) *Diagnosing the system for organizations*, Chichester West Sussex, New York: Wiley.

Bertalanffy, L.v. (1968) *General system theory : foundations, development, applications*, Rev. ed edn. New York : Braziller.

Beugnard, A., Jézéquel, J., Plouzeau, N. and Watkins, D. (1999) Making Components Contract Aware. *IEEE Computer* vol 32, no 7, pp 38-45.

Bracciali, A., Brogi, A., and Canal, C. (2002) Dynamically Adapting the Behaviour of Software Components. In Arbab, F. and Talcott, C., (Eds.) *Proceedings 5th International Conference on Coordination Models and Languages (Coordination'02) LNCS 2315* York, UK, Springer.

Bradbury, J.S. (2004) , Organizing Definitions and Formalisms of Dynamic Software Architectures. *Queen's University, Technical Report 2004-477*,

Bradbury, J.S., Cordy, J.R., Dingel, J., and Wermelinger, M. (2004) A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems* Newport Beach, California,

Bruneton, E., Coupaye, T., and Stefani, J.B. (2002) Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP02)* Malaga, Spain,

Bäumer, D., Riehle, D., Siberski, W. and Wulf, M. (2000) Role Object. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern languages of program design 4*, pp 15-32. Addison-Wesley

Cai, K., Cangussu, J.W., DeCarlo, R.A., and Mathur, A.P. (2004) An Overview of Software Cybernetics. In *Eleventh International Workshop on Software Technology and Engineering Practice* Amsterdan, The Netherlands, IEEE Computer Society.

Cangussu, J.W., Cooper, K., and Li, C. (2004) A Control Theory Based Framework for Dynamic Adaptable Systems. In *ACM Symposium on Applied Computing (SAC 2004)* Nicosia, Cyprus, ACM.

Chang, H. and Collet, P. (2005) Fine-grained Contract Negotiation for Hierarchical Software Components. In *Proceeding of 31st EUROMICRO Conference on Software Engineering and Advanced Applications,* IEEE.

Cheng, S.-W., Garlan, D., and Schmerl, B.R. (2005) Making Self-Adaptation an Engineering Reality. In Babaoglu et. al., (Ed.) *Self-star Properties in Complex Information Systems, Lecture Notes in Computer Science 3460* Bertinoro, Italy, Springer .

Cheng, S.-W., Huang, A.-C., Garlan, D., Schmerl, B., and Steenkiste, P. (2004) An Architecture for Coordinating Multiple Self-Management Systems. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)* Oslo, Norway, Kluwer Academic Publishers.

Collet, P. (2001) Functional and Non-Functional Contracts Support for Component-Oriented Programming. In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components, OOPSLA'2001* Tampa Bay (Florida),

Collet, P., Rousseau, R., Coupaye, T., and Rivierre, N. (2005) A Contracting System for Hierarchical Components. In *SIGSOFT Symposium on Component-Based Software Engineering (CBSE'05), LNCS 3489* St-Louis, Missouri, USA, Springer Verlag.

Colman, A. and Han, J. (2005a) Coordination Systems in Role-based Adaptive Software. In *Proceedings of the 7th International Conference on Coordination Models and Languages (COORD 2005), LNCS 3454* Namur, Belgium, LNCS 3454.

Colman, A. and Han, J. (2005b) On the autonomy of software entities and modes of organisation. In *Proceedings of the 1st International Workshop on Coordination and Organisation (CoOrg 2005)* Namur, Belgium,

Colman, A. and Han, J. (2005c) Operational management contracts for adaptive software organisation. In *Proceedings of the Australian Software Engineering Conference (ASWEC 2005)* Brisbane, Australia, IEEE.

Colman, A. and Han, J. (2005d) An organisational approach to building adaptive service-oriented systems. In Zirpins, C., Ortiz, G., Lamerdorf, W. and Emmerich, W., (Eds.) *Proceeding of First International Workshop on Engineering Service Compositions,WESC'05, in IBM Research Report RC23821* Amsterdam, The Netherlands, Yorktown Heights: IBM Research Division: IBM.

Colman, A. and Han, J. (2006a) Adaptive service-oriented systems: an organisational approach. *International Journal of Computer Systems Science & Engineering* vol 21, no 4, pp 235-246.

Colman, A. and Han, J. (2006b) Coordination Systems for Adaptive Software. Science of Computer Programming, Elsevier (Forthcoming)

Colman, A. and Han, J. (2006c) Using Associations Aspects to Implement Organisational Contracts. Electronic Notes in Theoretical Computer Science - Proceedings of the 1st International Workshop on Coordination and Organisation (CoOrg 2005) vol 150, no 3, pp 37-53.

Colman, A., Pham, L.D., Han, J., and Schneider, J.-G. (2006) Adaptive Application-Specific Middleware. In *Proceedings of the Middleware for Service Oriented Computing Workshop (MW4SOC)* Melbourne, Australia, ACM.

Coulson, G., Blair, G.S., and Grace, P. (2004) On the Performance of Reflective Systems Software. In *Proc. International Workshop on Middleware Performance (MP 2004)* Phoenix, Arizona,

Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004) OpenCOM v2: A Component Model for Building Systems Software. In *Proceedings of IASTED Software Engineering and Applications (SEA'04)* Cambridge, MA,

Cuesta, C.E., de la Fuente, P., and Barrio-Solárzano, M. (2001) Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM Symposium on Applied Computing SAC '01* Las Vegas, Nevada, United States, New York, NY: ACM Press.

de Lemos, R. and Fiadeiro, J.L. (2002) An architectural support for self-adaptive software for treating faults. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)* Charleston, SC, USA,

de Miguel, M.A. (2003) QoS modeling language for high quality systems. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003).*

Dennett, D.C. (1987) *The Intentional Stance*, Cambridge, Mass: MIT Press.

Dignum, V. (2003) A Model for Organizational Interaction, based onAgents, founded in Logic. PhD thesis, University of Utrecht.

Diotalevi, F. (2004) Contract Enforcement with AOP, *http://www-106.ibm.com/developerworks/library/j-ceaop/,* Last accessed: Sep 2004

Duce, D.A. (1991) *Workshop on User Interface Management Systems and Environments: User interface management and design proceedings of the Workshop on User Interface Management Systems and Environments, Lisbon, Portugal, June 4-6, 1990*, Berlin, New York: Springer-Verlag.

Eclipse Foundation (2004) AspectJ, *http://eclipse.org/aspectj/,* Last accessed: Oct 2004

Emmerich, W. (2000) *Engineering distributed objects*, Chichester : John Wiley & Sons.

Ferber, J. and Gutknecht, O. (1998) A Meta-Model for the Analysis and Design of

Organizations in Multi-Agent Systems. In *3rd International Conference on Multi-Agent Systems (ICMAS 1998)* Paris, France, IEEE Computer Society.

Fogel, D.B. and IEEE Neural Networks Council (1995) *Evolutionary computation toward a new philosophy of machine intelligence*, New York: IEEE Press.

The Foundation for Intelligent Physical Agents (2002) FIPA Communicative Act Library Specification, *http://www.fipa.org/specs/fipa00037/,* Last accessed: Aug 2004

Fowler, M. (1997) Dealing with Roles. In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs* Monticello, Illinois, USA, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University.

Gamma, E., Vlissides, J., Johnson, R. and Helm, R. (1995) *Design patterns elements of reusable object-oriented software*, Reading, Mass: Addison-Wesley.

Ganek, A.G. and Corbi, T.A. (2003) The dawning of the autonomic computing era. *IBM Systems Journal* vol 42, no 1, pp 5-18.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B. and Steenkiste, P. (2004) Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* vol 37, no 10, pp 46-54.

Garlan, D., Poladian, V., Schmerl, B., and Sousa, J.P. (2004) Task-based self-adaptation. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems* Newport Beach, California, ACM Press .

Garlan, D. and Shaw, M. (1993) An Introduction to Software Architecture. In: Ambriola, V. and Tortora, G., (Eds.) *Advances in Software Engineering and Knowledge Engineering*, pp 1-39. Singapore: World Scientific Publishing Company

Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (2002) The Belief-Desire-Intention Model of Agency. In *Proc. 5th Inter. Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*

Georgiadis, I., Magee, J., and Kramer, J. (2002) Self-organising software architectures for distributed systems. In *Proceedings of the First Workshop on Self-Healing Systems WOSS '02* Charleston, South Carolina, New York, NY: ACM Press .

Georgiadis, I. (2002) Self-Organising Distributed Component Software Architectures. PhD Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

Global Grid Forum (2004) Web Services Agreement Specification (WS-Agreement) www.gridforum.org/Meetings/GGF11/ Documents/draft-ggf-graap-agreement.pdf .

Gorlick, M.M. and Razouk, R.R. (1991) Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering* Austin, Texas, Los Alamitos, CA: Computer Society Press.

Hallsteinsen, S., Floch, J., and Stav, E. (2005) A Middleware Centric Approach to Building Self-Adapting Systems. In Gschwind, T. and Mascolo, C., (Eds.) *Prceedings of the 4th International Workshop Software Engineering and Middleware (SEM 2004), LNCS 3437* Linz, Austria, Springer-Verlag GmbH.

Han, J. (1998) A Comprehensive Interface Definition Framework for Software Components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference (APSEC'98)* Taipei, Taiwan, IEEE Computer Society Press.

Han, J. and Jin, Y. (2005) Runtime Validation of Behavioural Contracts for Component Software. In *Proceedings of the 5th International Conference on Quality Software (QSIC2005)* Melbourne, Australia, September 2005, IEEE Computer Society Press .

Hannemann, J. and Kiczales, G. (2002) Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)* ACM.

Helm, R., Holland, I., and Gangopadhyay, D. (1990) Contracts: specifying behavioral compositions in object-oriented systems . In *Proceedings of the European conference* *on object-oriented programming on Object-oriented programming systems, languages, and applications ECOOP '90* Ottawa, Canada , Publisher ACM Press New York, NY, USA .

Herring, C. and Kaplan, S. (2000) The Viable System Architecture. In *Thirty-Fourth Hawaii International Conference on System Sciences (HICSS-34)* Maui, Hawaii,

Herring, C.E. (2002) Viable software: The Intelligent control paradigm for adaptable and adaptive architecture, PhD Thesis. University of Queensland.

Herrmann, S. (2002) Object Teams: Improving Modularity for Crosscutting Collaboarations. In *Net.Object Days 2002* Erfurt, Germany,

Herrmann, S. (2005) Programming with Roles in ObjectTeams/Java. In *AAAI Fall Symposium, Roles, an interdisciplinary perspective* Arlington, Virginia, AAAI Press.

Heylighen, F. and Joslyn, C. (2001) Cybernetics and Second-order Cybernetics. In: Anonymous *Encyclopedia of physical science and technology*, 3 edn.

Hillman, J. and Warren, I. (2004) An Open Framework for Dynamic Reconfiguration. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*

Holland, J.H. (1992) *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*, Cambridge, Mass. : MIT press.

Holland, J.H. (1998) *Emergence : from chaos to order*, Oxford : Oxford University Press.

Horvitz, E. (1999) Principles of Mixed-Initiative User Interfaces. In *Proceeedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI 99)* Pittsburgh, Pennsylvania, USA, 159-166p.

Huang, A.-C. and Steenkiste, P. (2004) Building Self-configuring Services Using Service-specific Knowledge. In *Proceeding of 13th IEEE Symposium on High-Performance Distributed Computing (HPDC'04)* Honolulu, Hawaii, IEEE.

Huang, A.-C. and Steenkiste, P. (2005) Building Self-adapting Services Using Service-specific Knowledge. In *Fourteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-14)* Research Triangle Park, NC,

IBM Corporation (2003) Web Service Level Agreement (WSLA) Language Specification wsla-2003/01/28. IBM Corporation http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.

Jackson, D. (2002) Alloy: a lightweight object modelling notation. *ACM Transactions on Software Eng. Methodologies.* vol 11, no 2, pp 256-290.

Jin, Y. and Han, J. (2005) Runtime Validation of Behavioural Contracts for Component Software. In *Proceedings of the 5th International Conference on Quality Software (QSIC2005)* Melbourne, Australia, IEEE Computer Society Press.

Juan, T., Pearce, A. and Sterling, L. (2002) ROADMAP: extending the Gaia methodology for complex open systems. *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems, Bologna, Italy, ACM* pp 3-10.

KAOS (2003) Goal-Driven Requirements Engineering: the KAOS Approach, *http://www.info.ucl.ac.be/research/projects/AVL/ReqEng.html,* Last accessed: Feb 2004

Keeney, R.L. and Raiffa, H. (1976) *Decisions with multiple objectives : preferences and value tradeoffs*, New York : Wiley.

Kendall, E.A. (1999a) Role model designs and implementations with aspect-oriented programming. *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, Denver, CO* pp 353-369.

Kendall, E.A. (1999b) Role Modelling for Agents System Analysis, Design and Implementation. In *Proceeding of the 1st International Symposium on Agent Systems and Applications* 204-218p. Palm Springs (CA): IEEE CS Press.

Kephart, J.O. and Chess, D.M. (2003) The Vision of Autonomic Computing. *Computer, IEEE Computer Society Press Los Alamitos, CA, USA* vol 36, no 1, pp 41-50.

Khan, K. and Han, J. (2005) Deriving Systems Level Security Properties of Component Based Composite Systems . In *In Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)* Brisbane, Australia, IEEE Computer Society Press.

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C.V., Maeda, C. and Mendhekar, A. (1997) Aspect-oriented programming. In: Anonymous *European Conference on Object-Oriented Programming: ECOOP'97--object-oriented programming 11th European Conference, Jyväskylä, Finland, June 9-13,* Springer-Verlag

Knight, J.C., Heimbigner, D., Wolf, A., Carzaniga, A., Hill, J., Devanbu, P., and Gertz, M. (2002) The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In *Intrusion Tolerance Workshop, DSN-2002, The International Conference on Dependable Systems and Networks,* Washington DC,

Kokar, M.M., Baclawski, K. and Eracar, Y.A. (1999) Control Theory-Based Foundations of Self-Controlling Software. *IEEE Intelligent Systems* vol 14, no 3, pp 37-45.

Kristensen, B.B. and Osterbye, K. (1996) Roles: Conceptual Abstraction Theory & Practical Language Issues. *Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-Oriented Systems* vol 2, no 3, pp 143-160.

Kristensen, B.B. (1996) Object-oriented modeling with roles. In *Proceedings of the 2nd International Conference on Object-oriented Information Systems (OOIS'95)* Dublin, Ireland, Springer-Verlag.

Lee, J.S. and Bae, D.H. (2002) An enhanced role model for alleviating the role-binding anomaly. *Software: Practice and Experience* vol 32, pp 1317-1344.

Li, Z., Han, J. and Jin, Y. (2005) Pattern-Based Specification and Validation of Web Services Interaction Properties. *Lecture Notes in Computer Science, Springer* vol 3826: pp 73-86.

Lieberherr, K.J. (1996) *Adaptive object-oriented software the Demeter Method with propagation patterns*, Boston: PWS Pub. Co.

Ludwig, H., Dan, A., and Kearney, R. (2004) Cremona: an architecture and library for creation and monitoring of WS-agreements . In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)* New York, NY, USA,

Magee, J. and Kramer, J. (1996) Dynamic structure in software architectures. In D. Garlan, (Ed.) *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* San Francisco, California, New York, NY: ACM Press .

Maturana, H.R. and Varela, F.J. (1980) *Autopoiesis and cognition the realization of the living*, Dordrecht, Holland, Boston: D. Reidel Pub. Co.

Maturana, H.R. and Varela, F.J. (1987) *The tree of knowledge the biological roots of human understanding*, 1st ed edn. Boston: New Science Library. Distributed in the United State by Random House.

McKinley, P.K., Sadjadi, S.M., Kasten, E.P. and Cheng, B.H.C. (2004) Composing Adaptive Software. *IEEE Computer* vol 37, no 7, pp 56-64.

Medvidovic, N., Oreizy, P., Robbins, J.E., and Taylor, R.N. (1996) Using object-oriented typing to support architectural design in the C2 style. In D. Garlan, (Ed.) *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* San Francisco, California, New York, NY: ACM Press.

Medvidovic, N. and Taylor, R.N. (2000) A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering,* vol 26, no 1, pp 70-93.

Meyer, B. (1988) *Object-oriented software construction*, New York: Prentice-Hall.

Mintzberg, H. (1983) *Structure in fives: designing effective organizations*, Englewood-Cliffs, New Jersey: Prentice Hall.

Monroe, R. (2000) Capturing Software Architecture Design Expertise With Armani , Capturing Software Architecture Design Expertise With Armani. *CMU School of Computer Science Technical Report CMU-CS-98-163*,

Mukhija, A. and Glinz, M. (2003) CASA - A Contract-based Adaptive Software Architecture Framework. In *Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2003)* Berne, Switzerland,

Mukhija, A. and Glinz, M. (2005a) The CASA Approach to Autonomic Applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)* Paris, France,

Mukhija, A. and Glinz, M. (2005b) Runtime Adaptation of Applications through Dynamic Recomposition of Components. In *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS 2005)* Innsbruck, Austria,

Nagel, E. (1961) *The structure of science problems in the logic of scientific explanation*, New York: Harcourt, Brace & World.

Norman, D. (1984) Stages and levels in human-machine interaction. *International Journal of Man-Machine Studies* no 21, pp 365-375.

OASIS (2005) Web Services Distributed Management -Management of Web Services 1.0, OASIS Standard, 9 March 2005 wsdm-mows-1.0http://docs.oasis-open.org/wsdm/2004/12/cd-wsdm-mows-1.0.pdf.

Object Management Group (2004) UML 2.0 Superstructure (Final Adopted specification), *http://www.uml.org/#UML2.0,* Last accessed: Oct 2004

Odell, J., Nodine, M. and Levy, R. (2005) A Metamodel for Agents, Roles, and Groups. *Agent-Oriented Software Engineering (AOSE) V*

Odell, J., Parunak, H.V.D., Brueckner, S. and Fleischer, M. (2004) Temporal Aspects of Dynamic Role Assignment. *Agent-Oriented Software Engineering (AOSE) IV, Lecture Notes on Computer Science , Springer, Berlin* **2935** 201-213.

Odell, J., Parunak, H.V.D., Brueckner, S. and Sauter, J. (2003) Changing Roles: Dynamic Role Assignment. *Journal of Object Technology, ETH Zurich* vol 2, no 5, pp 77-86.

Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S. and Wolf, A.L. (1999) An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* vol 14, no 3, pp 54-62.

Parnas, D.L. (1972) On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* vol 15, no 12, pp 1053-1058.

Parunak, V. and Brueckner, S. (2003) Engineering Self-Organising Applications. *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent System (AAMAS'03)* pp 1-54.

Pham, L.D., Colman, A. and Han, J. (2005) The implementation of message synchonisation, queuing and allocation in the ROAD framework *Technical Report SUT.CeCSES-TR009*, Faculty of ICT, Swinburne University of Technology.

Plasil, F. and Visnovsky, S. (2002) Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering , IEEE Press* vol 28, no 11, pp 1056-1076.

Poladian, V., Sousa, J.P., Garlan, D., and Shaw, M. (2004) Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)* Edinburgh, UK,

Rajan, H. and Sullivan, K. (2003) Eos:instance-level aspects for integrated system design. *ACM SIGSOFT Software Engineering Notes* vol 28 , no 5 , pp 297-306.

Rasmussen, J., Pejtersen, A.M. and Goodstein, L.P. (1994) *Cognitive systems engineering*, New York: Wiley.

Reenskaug, T. (1996) *Working with Objects : the OOram Software Engineering Method*, Manning Publications Co.

Riehle, D. (1998) Bureaucracy. In: Martin, R., Riehle, D. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*, pp 163-186. Reading, Massachusetts: Addison-Wesley

Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., and Komiya, S. (2004) Association Aspects. In *Proceedings of the Aspect-Oriented Software Development '04* Lancaster U.K, ACM.

Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S. and Komiya, S. (2006) Design and Implementation of an Aspect Instantiation Mechanism . *Transactions on Aspect-Oriented Software Development, LNCS* vol 3880, pp 259-292.

Searle, J.R. (1983) *Intentionality, an essay in the philosophy of mind*, Cambridge Cambridgeshire, New York: Cambridge University Press.

Shannon, C.E. and Weaver, W. (1949) *The mathematical theory of communication*, Urbana Ill. : University of Illinois Press.

Shaw, M. (1995) Beyond objects: A software design paradigm based on process control. *ACM Software Engineering Notes* vol 20, no 1, pp 27-39.

Shaw, M. and Garlan, D. (1996) *Software architecture perspectives on an emerging discipline*, Upper Saddle River, N.J: Prentice Hall.

Shoham, Y. and Tennenholtz, M. (1995) On social laws for artificial agent societies: off-line design. *Artificial Intelligence* vol 73, no 1-2, pp 231-235.

Sichman, J., Dignum, V. and Castelfranchi, C. (2005) Agent Organizations: a Concise Overview. *Special Issue in Agent Organizations of Journal of the Brazilian Computer Society* vol 11, no 1, pp 3-8.

Simon, H.A. (1957) *Administrative behavior : a study of decision making process in administrative organization*, 2nd ed. with a new introd edn. New York : Macmillan.

Simon, H.A. (1969) *The sciences of the artificial*, Cambridge: M.I.T. Press.

Skene, J., Lamanna, D.D., and Emmerich, W. (2004) Precise Service Level Agreements. In *26th International Conference on Software Engineering (ICSE'04)*

Skyttner, L. (2001) *General systems theory ideas & applications*, Singapore, River Edge, N.J: World Scientific.

Sousa, J.P. and Garlan, D. (2003) The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *CMU-CS-03-183*, School of Computer Science, Carnegie Mellon University. www.cs.cmu.edu/~jpsousa/research/CMU-CS-03-183.pdf .

Steimann, F. (2000) On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering* no 35, pp 83-106.

Steimann, F. (2005) The Role Data Model Revisited. In *AAAI Fall Symposium, Roles, an interdisciplinary perspective* Arlington, Virginia, AAAI Press.

Sullivan, J.W. and Tyler, S.W. (1991) *Intelligent user interfaces*, New York, N.Y, Reading, Mass: ACM Press. Addison-Wesley Pub. Co.

Sullivan, K., Gu, L., and Cai, Y. (2002) Non-modularity in aspect-oriented languages: integration as a crosscutting concern for *AspectJ.* In *Proceedings of the 1st international conference on Aspect-oriented software development, AOSD 02* Enschede, The Netherlands, ACM Press.

Sykes, J.A. (2003) Negotiating early reuse of components: a model-based analysis. In: Favre, L., (Ed.) *UML and the unified process*, pp 66-69. Hershey, PA, USA: Idea Group Publishing

Szyperski, C. (1997) *Component software : beyond object-oriented programming*, New York :, Harlow, England ;, Reading, Mass. : ACM Press ; Addison-Wesley.

Taylor, M.M. (1988) Layered Protocols for computer-human dialogue. I: Principles. *International Journal of Man-Machine Studies* no 28, pp 175-218.

Tosic, V. and Pagurek, B. (2005) On comprehensive contractual descriptions of Web services. In *Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service. EEE '05.*

van Lamsweerde, A. (2001) Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings. RE'01 - International Joint Conference on Requirements Engineering, Toronto, IEEE,* pp 249-263.

van Lamsweerde, A. (2003) From System Goals to Software Architecture. In: Bernardo, M. and Inveradi, P., (Eds.) *Formal Methods for Software Architecture - LNCS 2804*, Springer-Verlag

W3C (2004) WS Choreography Model Overview WD-ws-chor-model-20040324http://www.w3.org/TR/2004/WD-ws-chor-model.

W3C (2005) Web Services Description Language (WSDL) Version 2.0 Primer WD-wsdl20-primer-20050510 http://www.w3.org/TR/wsdl20-primer/.

Waewsawangwong, P. (2004) A Constraint Architectural Description Approach to Self-Organising Component-Based Software Systems. In *26th International Conference on Software Engineering (ICSE'04)* Edinburgh, United Kingdom, IEEE Computer Society.

Walsh, W.E., Tesauro, G., Kephart, J.O., and Das, R. (2004) Utility Functions in Autonomic Systems. In *1st International Conference on Autonomic Computing (ICAC 2004)* New York, NY, USA, IEEE Computer Society.

Wermelinger, M. (1998) Towards a chemical model for software architecture reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems* IEEE Computer Society Press.

Wermelinger, M., Fiadeiro, J.L., Andrade, L., Koutsoukos, G., and Gouveia, J. (2001) Separation of Core Concerns: Computation, Coordination, and Configuration. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems* Tampa Bay, FL,

Wiener, N. (1961) *Cybernetics, or, Control and communication in the animal and the machine*, 2nd ed edn. New York : M.I.T. Press.

Wirfs-Brock, R. and McKean, A. (2002) *Object Design: Roles, Responsibilities, and Collaborations*, Addison Wesley.

Wooldridge, M.J. and Jennings, N.R. (2000) The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems* vol 3, no May - June, pp 285-315.

Yellin, D.M. and Strom, R.E. (1997) Protocol specifications and component adaptors . *ACM Transactions on Programming Languages and Systems (TOPLAS), ACM Press New York, NY, USA* vol 19, no 2, pp 292-333.

Zambonelli, F., Jennings, N.R., and Wooldridge, M.J. (2000) Organisational Abstractions for the Analysis and Design of Multi-Agent Systems. In *Workshop on Agent-oriented Software Engineering ICSE 2000*

Zambonelli, F., Jennings, N.R. and Wooldridge, M. (2003) Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)* vol 12, no 3, pp 317-370.

Zhou, Z. and McKinley, P.K. (2005) COCA: A Contract-Based Infrastructure for Collaborative Quality-of-Service Adaptation. *Technical Report MSU-CSE-05-20*, Computer Science and Engineering, Michigan State University, East Lansing, Michigan.

Zirpins, C., Lamersdorf , W., and Baier, T. (2004) Flexible coordination of service interaction patterns . In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)* New York, NY, USA , ACM Press.

.