Coordination Systems in Role-Based Adaptive Software

Alan Colman and Jun Han

Faculty of Information and Communication Technologies,
Swinburne University of Technology,
Melbourne, Victoria, Australia
{acolman,jhan}@swin.edu.au

Abstract. Software systems are becoming more open, distributed, pervasive, and connected. In such systems, the relationships between loosely-coupled application elements become non-deterministic. Coordination can be viewed as a way of making such loosely coupled systems more adaptable. In this paper we show how coordination-systems, which are analogous to nervous systems, can be defined independently from the functional systems they regulate. Such coordination systems are a network of organisers and contracts. We show how the contracts that make up the coordination-system can be used to monitor, regulate and configure the interactions between clusters of software entities called roles. Management and functional levels of contracts are defined. Management contracts regulate the flow of control through the roles. Functional contracts allow the specification of performance conditions. These contracts bind clusters of roles into self-managed composites — each composite with its own organiser role. The organiser roles can control, create, abrogate and reassign contracts. Adaptive systems are built from a recursive structure of such self-managed composites. The network of organiser roles and the contracts they control constitute a coordination-system that is a *separate concern* to the functional system. Association aspects are suggested as a mechanism to implement such coordination-systems.

1 Introduction

As software systems become more open, distributed, pervasive, and connected, such systems need to be able to adapt to their dynamic environments. One approach to building adaptable and adaptive systems is to construct them of loosely coupled elements. These elements are dynamically coordinated to meet changing goals and environmental demands. This paper proposes that coordination functions be implemented as a separate sub-system. This coordination-system can be described and controlled independently from the sub-systems 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 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

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 63 – 78, 2005.
© Springer-Verlag Berlin Heidelberg 2005

abstraction from the functional processes that transform labour and material into products.

The aim of this paper is to show how such coordination/control systems can be imposed on top of functional software, to better monitor, regulate, and coordinate those systems in dynamic environments. The structure of these coordination-systems can be defined as a *separate concern* from the functional systems they coordinate.

Coordination-systems have a mapping to the state of the underlying functional system. The type(s) of abstraction on which the coordination-system is built will depend on the variables that need to be controlled in order to maintain the system's viability in its environment. In terms of a biological analogy, a controlled variable is the level of oxygen supply to the cells. In a management system, the variable might be the amount of funds in the bank. In computerised systems, such control variables could be computational or communication resources; or environmental variables.

This paper is structured as follows: Section 2 gives an overview of a schema for coordination systems based on roles, along with a motivating example. Section 3 examines *contracts* between roles, and the organiser-roles that create, monitor and control contracts. In Section 4 we introduce the concept of adaptive coordination systems built from these organiser roles and contracts. Coordination systems enable the functional systems to maintain their viability in dynamic environments and to respond to changes in non-functional requirements (NFRs). Section 5 discusses the implementation of coordination systems using association aspects. Section 6 briefly looks at related work and Section 7 concludes.

2 Coordination as the Control of Interactions Between Roles

The ROAD (Role-oriented adaptive design) framework presented in this paper extends work on object-oriented role and associative modelling in [1-4]. In this framework, the elements that are being coordinated are *roles* played by *objects*. A role satisfies responsibilities to the system as a whole. Roles are first-class entities that can be added to, and removed from, objects. Kristensen [3] provides a definition of roles that is based on the distinction between intrinsic and extrinsic members (methods and data) of an object. Intrinsic members provide the *core* functionality of the object, while extrinsic members contain the functionality of the role. In our view, this 'core functionality' is the situated computational and communication *capabilities* of the object.

The ROAD approach to creating adaptive software systems is based on the distinction between two types of role – *functional* roles and *management* roles. *Functional roles* (or more properly *domain*-function roles) are focused on first-order goals — on achieving the desired application-domain output. Functional roles constitute the *process* as opposed to the *control* of the system. In ROAD these functional roles are decoupled; they do not directly reference each other. Functional roles are associated by contracts that mediate the interactions between them. The creation and monitoring of these contracts is the responsibility of a type of management role — *organiser-role*s.

To help us discuss the coordination of roles we will consider an example of a highly simplified business department that makes Widgets and employs Employees with different skills to make them. In such a business organisation an employee can perform a number of roles, sometimes simultaneously. Employees (objects) can perform the roles of 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 WidgetDept Organiser role is responsible for creating the bindings between roles and the objects that play them, and for creating the associations between the various functional roles.

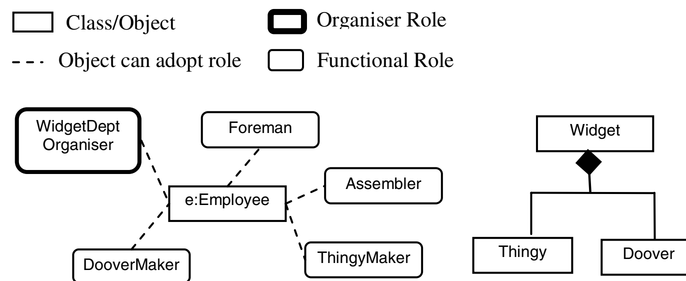

Fig. 1. Roles and objects in a Widget Department

In a purely object-oriented analysis only the domain function is modelled. For example, in the above department the various roles might be modelled as sub-class specialisations of the Employee class. However, in an *open* system we cannot assume that all objects of type Employee have the same *capabilities*. In a real-world manufacturing department some ThingyMakers would be more capable than others (e.g. faster, more accurate, more adaptable etc.). In the ROAD schema, we identify this difference by separating roles from objects, where the situated objects provide the performance capability to the purely domain-function roles. Object-role pairs may be running in different computational and communication contexts. These contexts affect the relative performance of roles (e.g. they may be faster, more reliable, better connected, more costly etc.). As well as the variation in computational contexts, the relationship between the Widget department and its environment may also vary. For example, orders flowing into the department to make new widgets might exceed the capacity of the department to manufacture them. The organiser role may have to reconfigure the relationships between the functional roles, or else create extra object-roles in order to meet the increased demand.

In the Fig. 2 below, an organiser-role (CR) creates, monitors and controls the contracts (C1, C2) between functional roles (F1, F2, F3). The domain of organiser-roles is the system itself rather than the problem-domain. Each organiser role is responsible for a cluster of functional roles. We will call these regulated clusters of roles "self-managed composites" because each composite attempts to maintain a homeostatic relationship with its environment and other composites. We use the word

composite rather than *component* because the roles in the composite are not necessarily encapsulated in a single package (although they may be). 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 perform a definable domain function, and can themselves be part of higher-level composites. 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 the *coordination-system*. An example schema of a coordination-system's topology is illustrated in Fig. 2 below.

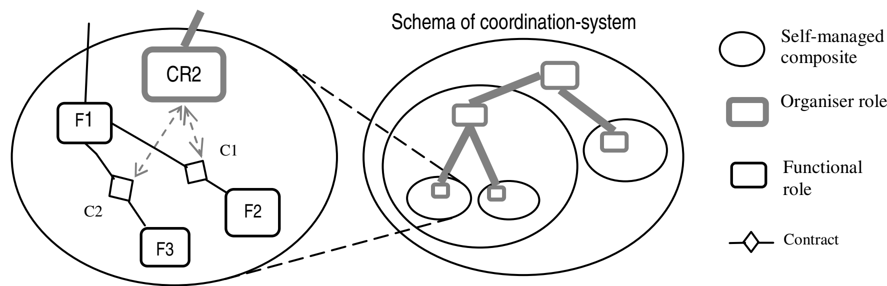

Fig. 2. Self-managed Composite with a Coordination-system

In following sections we show how a coordination-system, built from contracts and organisers, can be imposed on a cluster of functional roles to enable the system to better cope with uncertain environments and changing NFRs.

3 Controlling Interactions Using Contracts

In the first part of this section we summarise our previous work [5] on using contracts to manage interactions between roles. This approach distinguishes contracts at the functional and management level. In the latter part of this section, we extend this work by introducing the concept of management-level protocols that control interaction between roles, and then discuss what is needed to specify contracts at the functional level.

ROAD contracts [5] are association classes that express the obligations of the contract parties to each other. The *form* (type) of a contract sets out the mutual obligations and interactions between classes of party (e.g. vendor and purchaser). A contract is *instantiated* when values are put against the variables in the contract *schedule* (e.g. the vendor and purchaser are named, the date of commencement is agreed etc.) and the contract is signed. Contracts also have an *execution state* that indicates the level of fulfilment of the various clauses of the contract. This dynamic

information is needed to ensure that the terms of the contract are being met. It includes information on the state of the relationship between the parties, and the state of any interaction defined by the protocols. ROAD contracts store this dynamic state information in the contract itself. The contract itself can enforce the terms of the contract by controlling the interactions between the parties.

Contracts between functional roles often share common characteristics. In particular, the *control* aspects of the contract can be abstracted. Using our example of a WidgetMaking department, the control-management relationship between a Foreman and an Assembler could be characterised as a Supervisor-Subordinate relationship. Similarly, a Foreman-ThingyMaker relationship would also be characterised a Supervisor-Subordinate relationship. Rules control the interactions between *operational-management* roles such as a supervisor and a subordinate. For example a supervisor can tell a subordinate to do a work related task but a subordinate *cannot* tell a supervisor what to do. Other types of management contract include auditor – auditee; peer – peer; and predecessor – successor in supply-chain and production-lines.

In ROAD, these abstracted control aspects of the relationships between roles are encapsulated in a *management contract*. Domain-function-level contracts ("functional contracts" for short) inherit control relationships from these management contracts. The conceptual relationships between functional and management contracts, and the respective roles they associate, are illustrated in Fig. 3 below.

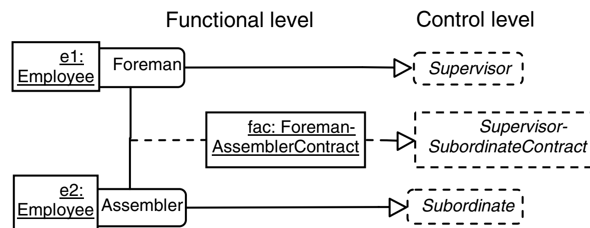

Fig. 3. Functional and management contracts

As can been seen from Fig. 3 above, `fac` is an *instance* of a `Foreman-AssemblerContract`. The contract 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 plays a Supervisor role in relation to the Assembler's Subordinate role.

Fig. 4 below illustrates an organisational structure for our Widget department, based on the Bureaucracy pattern [6] and built using contracts. In order to simplify the diagram, functional contracts have been drawn as diamonds. The structure, which is similar to a business organisation chart, is still abstract because objects have not yet been assigned to roles. Note that the Foreman plays *both* Supervisor and Subordinate roles in the organisational structure.

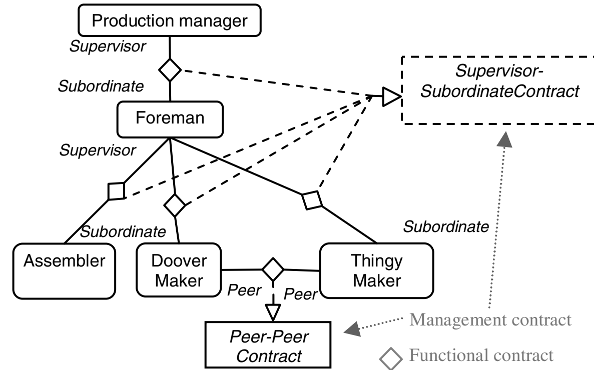

Fig. 4. Domain specific abstract organisation bound by contracts

In the next two sections we look at management and functional contracts in more detail. *Management contracts* specify the type of communication acts and protocols that are permissible between the two parties. *Functional contracts* specify the performance obligations of each party to the other.

3.1 Contracts at the Management Level

Contracts at the management level of abstraction *restrict* the types of method that roles can invoke in each other. From our example above, the Supervisor-Subordinate operational-management role association restricts interactions between the objects playing the ThingyMaker and the Foreman to certain *types* of interaction. For example, a ThingyMaker cannot tell its Foreman Supervisor what to do. Furthermore, these contracts only *allow* interactions between particular instances of object-roles. For example, the method `ThingyMaker.setProductionTarget()` can only be invoked by the ThingyMaker's own Foreman.

Control-Communication Acts. The control communication in management contracts can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the *control* aspects of the communication from the functional aspects. In [5], we defined a simple set of CCAs for direct and indirect control and for information passing. As an example, Table 1 below defines a set of primitives suitable to a hierarchical organisatiop. Unlike the communication act primitives in agent communication languages such as FIPA-ACL [7], CCAs express only *control* relationships between the two parties bound in a contract, rather than the types of communication between two independent agents. Note that indirect CCAs carry a reference to a resource `r`. The set below is not logically complete. For instance, it does not capture a referential command relationship (A tells B to tell C to do something), but it is sufficient to allow us to define a number of contracts between operational-management roles. From these contracts we can create organisational structures.

Table 1. Example of Control-Communication Act Primitives

Type of communication	Communication control acts
Direct Control	DO, SET_GOAL
Indirect Control	RESOURCE_ALLOCATE(r), RESOURCE_REQUEST(r)
Information	ACCEPT, REJECT, INFORM, QUERY, ACKNOWLEDGE

In terms of the primitives we defined above, Supervisors can initiate some types of interaction and Subordinates others. Initial CCAs for these roles are:

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

Other forms of management contract such as Peer-Peer would have different sets of valid initial CCAs for each party.

CCA Protocols. Protocols can be defined from CCA primitives. These Control Protocols (CPs) are sequence patterns of CCAs that terminate when the task is achieved. They 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 content of the task.

There are a limited number of protocols that form sensible interactions. It is possible to represent these protocols as strings of CCAs between initiator and respondent. To do this we will encode the CCAs as single letters so that complete protocols can be represented as individual strings. The codes for the above CCAs (plus “no response”) are defined as follows:

Table 2. CCA short hand codes

D	DO	I	INFORM	X	NO RESPONSE
G	SETGOAL	Q	QUERY	K	ACKNOWLEDGE
L	RESOURCE_ALLOCATE	A	ACCEPT		
S	RESOURCE_REQUEST	R	REJECT		

For further clarity we can apply the convention that initiator CCAs are capitalised, and respondent CCAs are in lower case. For example, the string “`Da`” indicates that the initiating party asks the respondent to do something, and that the respondent subsequently accepts. Control protocol clauses can also be represented by finite state machines (FSMs). These FSMs keep track of the conversation between two contracted parties. Clauses can have as a goal the *maintenance* of a state or the *achievement* of a state. In the case of maintenance clauses, successful transactions of the protocol will result in a return to a ‘ready’ state. The successful completion of achievement clauses will result in a ‘completed’ state for that clause. Where an interaction is following a protocol in which a response is expected but not forthcoming after a specified time, the protocol may specify that `n` retires are permitted before the clause is violated. The nodes in Fig. 5 below represent CCAs issued by either the initiator or the respondent in the transaction of a particular clause

in the contract. The letters in the nodes are a short hand for CCAs, as defined in Table 2 above (initiator CCAs in capitals, respondent CCAs in lower case). The FSM for the Supervisor-Subordinate contract *valid* protocols starting with a Supervisor D is illustrated below.

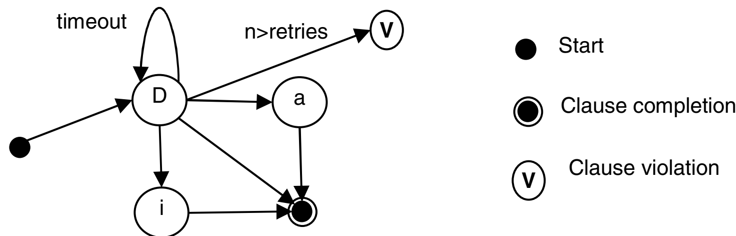

Fig. 5. Valid CCA protocol initiated with a Supervisor DO CCA

The form of our Supervisor-Subordinate management contract has been summarised in Table 3 below. The valid protocol sequences are expressed as strings as defined above. The management contract clause will be violated if the sequence of interactions does not follow one of these strings. For each protocol clause where a response is expected, values for the response timeout and the number of permissible retires would need to be specified (this is done at the functional level of the contract). Other types of management contract will have different sets of permissible protocols. For example, the protocol "`DaR`" is presumably acceptable in a Peer-Peer management contract where the "a" is a conditional accept. If management contracts are to enforce CCA protocols, 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.

Table 3. Example form of an operational-management contract

Management Contract	
Name	`Supervisor-Subordinate`
Party A	`Supervisor`
Party B	`Subordinate`
A initiated clauses	`D; Da; Di; I; Ik; Qi; G; Ga; Gi; L; La; ...`
B initiated clauses	`I; Ik; Qi; Sa; Sr; Si ; ...`

Management contracts are very limited in that they only monitor or enforce the *form* of the communication between the parties — there is no domain content apparent at the management control level of abstraction. Values for timeouts (in the event of no response), and values for the number of retries that are permitted, only make practical sense in relation to a domain specific function. Such values, along with the identity of the parties to the contract and other performance requirements for clauses and the contract as a whole, are provided by functional contracts. *Functional*

contracts specialise abstract *management* contracts. It is the functional contract rather than the management contract that is instantiated. When a functional role in an organisational structure is bound to an operational-management role using such a contact, all functional role method invocations and responses between the parties are associated with CCA primitives.

3.2 Contracts at the Functional Level

Contracts, at the abstraction level of domain function (functional contracts), add schedule details to the form of contract defined by its management contract. They also define extra clauses that are needed to govern the domain-specific interactions between the functional roles. Instances of functional contracts, such as a Foreman-ThingyMaker contract, include a completed contract schedule as well as clauses relating to protocols, performance conditions and contract breach. They are specified as follows:

Schedule details such as the names of the parties to the contract and any temporal constraints on the contract – commencement date, duration etc – are specified. The operational-management roles in the management contract are specialised with functional roles. For example, the supervisor party is specialised as a foreman.

The rules for communication that are defined in the management contract protocols are made concrete. CCAs are mapped to the method signatures in the functional roles (see the section below on implementation). For example, the method `do_make-Thingy()` of the ThingyMaker `tm` would be matched to the DO CCA. This means a supervisor, contracted to `tm`, can invoke this method. If a *control* protocol is to be enforced for an interaction, a valid protocol string is assigned to the interaction, and values are specified for retries and timeouts. As long as the protocol is followed during the interaction, the contract clause will not be violated. Protocol sequences of particular *domain-function* interactions which need to be followed by the parties [8]might also be specified, as in [9]. 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.

Clauses relating to any preconditions, post-conditions and invariants for the interactions are specified. These conditions are similar to those defined in the design-by-contract (DBC) approach [10], where such conditions are aimed at ensuring the correct functioning of the software. Consequently, these DBC conditions themselves cannot be changed. In functional-level contracts these conditions *can* be varied (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. Any performance conditions also are specified. For example, a foreman might specify the maximum time allowed for a contracted thingyMaker to produce a thingy. These NFRs reside in, and are enforced by, the contract rather than the component itself.

Finally, clauses relating to clause violation and what constitutes a breach of contract are specified. Some clause 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. Other clauses may not be as critical to the contract and will be monitored. The contract contains metrics to measure the performance of the clauses: for example ‘average time to make a widget’. Such metrics are monitored by the organiser role, which may then choose to abrogate the contract if the performance is unacceptable – and, for example, if another role-object is available to perform the function. There may also be remedies for clause violation. If a clause is violated it may be permissible to renegotiate the contract e.g. to a different service level.

During execution, a contract itself monitors the interactions between the roles. The contract will prevent unauthorised or invalid interactions and monitor all interactions in order to maintain the state of execution of its clauses. The contract also keeps the state of any performance metrics updated. 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 section.

4 The Coordination-System

A coordination-system is constructed of contracts and the organisers that control them. Every organiser role is responsible for a cluster of roles and contracts. Organisers have three main functions: Firstly, organisers control and monitor their contracts. An organiser can instantiate, change, abrogate and reassign its contracts. Organisers monitor the contracts in their self-managed composite for under-performance. Secondly, organisers can reconfigure the contracts to try to remediate any underperformance that results from perturbations or changing requirements. Any such reorganisation must maintain the composite’s viability. For example, if the structure is based on a Bureaucracy pattern [6], the organiser must ensure proper chains-of-responsibility (i.e. supervisor-subordinate chains) are maintained to preserve the functional flow-of-control. The organiser also creates role-object bindings [11] (the discussion of such bindings is beyond the scope of this paper).

Thirdly, organisers are the nodes of the coordination-system network. They interpret regulatory control messages that flow through this network and translate these messages into contract clauses. The coordination-system is a hierarchy in which non-functional performance requirements flow down, and information on the performance of the managed composites flow up. We will call these two regulatory control-message flows, respectively, *performance/constraint-propagation* and *performance-monitoring*.

The structure and adaptive behaviour of a coordination-system will be illustrated by looking at a ThingyMaking team within our Widget making department (WMD). 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.

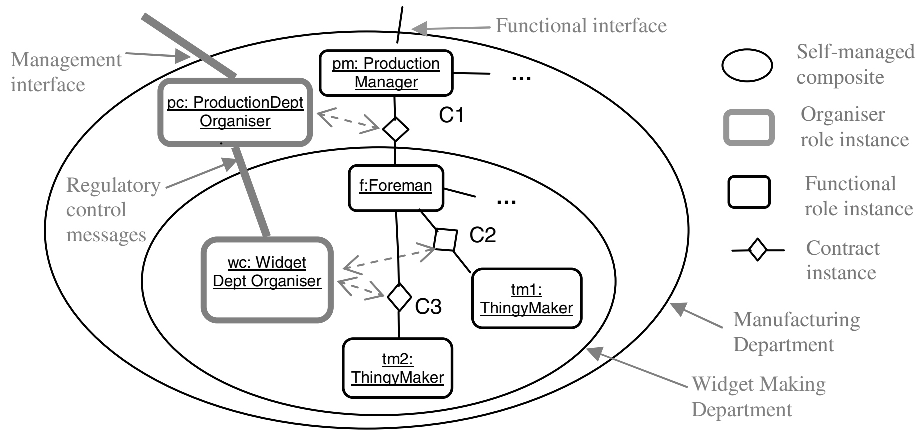

Fig. 6. Role-instance diagram of self-managed composites

Performance and Constraint Propagation. Performance requirements pass down the hierarchy of organiser roles to alter the performance requirements of the contracts. In our Manufacturing department the Production Manager receives (from above) orders for Widgets. It determines the priority of the orders, and passes these on to the Foreman (as determined by the contract `C1`). To fulfil its obligations under the `C1` contract, the foreman must organise the production of thingy orders within a specified timeframe. For example, the management level of contract `C1` allows the Production-Manager pm to invoke the Foreman `do_thingyOrder(…)` method. The contract `C1`, with additional advice it receives from the organiser role (`pm`), can add performance requirements and constraints. For example, the contract may require that thingies be made within certain time constraints, or that certain resource costs not be exceeded.

The Foreman `f` in turn allocates work to, among others, the thingyMakers (`tm1` and `tm2`). The Foreman can do this under the terms of the Foreman-ThingyMaker contract (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 object attached to `tm1` can make 10 thingies per hour while the role-player object attached to `tm2` can only make 5 thingies per hour. Because the Foreman f is party to contracts `C2` and `C3`, it can use the performance capacity information in the schedule in deciding to whom the work should be allocated.

Performance and constraint information can be about either the *actual* capacity or the *required* capacity. The *actual*-performance and constraint information is held by the contracts themselves. For example, contracts `C2` and `C3` contain information on the respective thingy-making-abilities of their contracted thingyMaker role-objects. The source of this actual-performance information could be from a specification provided by the builder of the component/system, or it could be derived from the

organiser role (e.g. the WidgetDeptOrganiser `wc`) monitoring the situated performance of the role-objects (see the next section).

Required-performance information on goals and constraints is transmitted through the organiser roles. The organiser role (e.g. `wc`) receives NFRs from the organiser above it in the coordination hierarchy (e.g. `pc`), then interprets these into performance requirements for the contracts it controls.

Performance Monitoring and Breach Escalation. In addition to its responsibility for characterising the actual-performance of role-objects mentioned above, 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* capacity (as expressed in the contracts) falls below the *required* capacity, then the organiser must attempt to reconfigure the composite by altering the existing contracts, reassigning roles to more capable objects or creating new contracts and roles.

If this reconfiguration is beyond the capability of the composite organiser, the organiser must inform the role above it in the coordination system hierarchy. Contract breach can occur if the violation(s) is severe enough. If a breach occurs (or if the organiser has the intelligence to predict a breach from the actual and required performance information), the composite organiser needs to reconfigure the contracts and roles. In the figure above, if wc detects the thingyMakers it controls are, or will be, over-loaded, it may ask `pc` for resources to get more thingyMakers.

Such escalation can be viewed as an organised form of exception-handling where control-messages flow up through the *coordination*-system, before error-messages flow back up the *functional*-system's flow of control. Just as an animal detecting a threat will increase its adrenaline levels to stimulate the heart rate for flight or fight, so the coordination- system detects stress on the system then changes the parameters of contracts (or reorganises them) in an attempt to avert system failure.

The performance parameters that an organiser could monitor include the rate of the flow of data resources through the system, the state of communication networks or the computational loads on the objects performing the various roles. The organiser's ability to successfully reconfigure the composite will depend on its ability to sense the performance parameters from either the contracts or the environment, its ability to reason about the causes of underperformance, and its ability to implement effective re-organisations. The discussion of such adaptive feedback loops is beyond the scope of this paper.

5 Implementing Coordination-Systems Using Association Aspects

If a coordination-system is to be implemented as a separate concern, the contracts and organiser roles that constitute the system must be able to be defined separately both at conceptual and code levels. A coordination-system *cross-cuts* the program structure defined by the functional-roles and classes. This section briefly describes how we have implemented contracts using association aspects. A more comprehensive description can be found in [5] and [12].

Aspect-oriented methods and languages seek to maintain the modularity of separate cross-cutting concerns in the design and program structures. The AspectJ [13] extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow). 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.

Implementing contracts as aspects enables interactions to be monitored and controlled. As implemented in [5], method signatures follow an arbitrary naming convention that indicate their CCA type (e.g DO CAA method names start with `do_`). At compile time, CCA pointcuts are pattern matched against methods so that contract advice can be woven into the functional code. Such advice can prohibit or permit interaction. Advice can be inserted before and after method invocations, allowing the state of the contract to be updated and the performance of the contract to be measured. Pointcuts also allow the execution context of the invoking and target objects to be exposed. Thus the state of the objects that participate in the contracts can also be accessed. This enables pre and post conditions to be monitored and enforced.

Aspects, as currently implemented in AspectJ, do not easily represent the behavioural associations between objects [14]. Current implementations of AspectJ provide *per-object* aspects. These can be used 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. This allows the aspect to maintain a unique state for each object, but *not* for associations of objects.

Sakurai et al. [15] have implemented an extension to the AspectJ compiler to handle *association-aspects.* Association-aspects are declared with a *`perobjects`* modifier that takes, as an argument, a tuple of objects. Instances of these association-aspects are suitable for implementing management and functional contracts. A contract can be created as an aspect instance that associates a group of objects. In previous work [5], we have demonstrated the implementation of contracts that control the communication between roles. Due to space constraints here, our notated Java code that demonstrates the creation, revocation, and reassignment of contracts is available at [12]. The example code at [12] also shows how to intercept various CCA-type method-calls between various authorised and unauthorised parties; how to permit, modify or prohibit the execution of the interaction; and how to keep a contract FSM updated.

The use of aspects to implement contracts allows functional code to be developed independently from the contracts that associate them. The only dependency in the functional code is that method signatures need to follow a CCA naming convention. However, there is a limitation in using AspectJ to create contracts: both the coordination code and the functional code must be compiled together in order for weaving to occur. This means new classes cannot be added dynamically without recompilation. Other approaches, such as load-time weaving of byte-code, might prove effective in addressing this limitation.

6 Related Work

ROAD extends work on role and associative modelling in [1-4,16]. Kendall [2] has shown how aspect-oriented approaches can be used to introduce role-behaviour to objects. Roles are encapsulated in aspects that are woven into the class structure. While these role-oriented approaches decouple the class structure, they do not explicitly define a coordination-system using management contracts. They are primarily concerned with role-object bindings rather than role associations.

The coordination model outlined here adopts a control-oriented [17] architectural approach, primarily focused on adaptivity rather than synchronisation. It has many similarities and some major differences with work by Andrade, Wermelinger and colleagues [18-20]. Both approaches represent contracts as first-class entities, and both use a layered architecture. In [19,20] the layers are Computation, Coordination and Configuration ('3C'). This is broadly similar to ROAD's four layer architecture (Computational-object, Functional-role, Management-contract, Organisation) with 3C's Computation layer similar to ROAD's Object and Functional role layers. 3C's contracts are method-centric rather than role-association-centric. They define a single interaction sequence that might involve many parties, whereas ROAD contracts are currently limited to two roles and many involve many types of interaction. Both approaches use contracts to model unstable aspects of the system, but 3C's focus is on business rules whereas ROAD focuses on performance variability. In 3C, there is no concept of a coordination network through which regulatory control messages pass.

The concept of a CCA in this paper is derived from the concept of a *communication act* in agent communication languages such as FIPA-ACL [7]. CCAs, as defined here, are much more restricted in their extent. CCAs deal only with control communication, and do not have to take intentionality of the other parties into account [21]. Work on roles has also been undertaken in multi-agent systems (MAS) [22-24]. In particular, [21] extends the concept of a role model to an organisational model. MAS systems, however, rely on components that have deliberative capability and more autonomy than the objects and roles discussed here. These agents negotiate interactions with other agents to achieve system level goals. These negotiations occur within a more amorphous structure than is defined here.

Like our approach, control-theoretic architectures separate control from functional processes [21]. Such systems are designed to maintain system viability during anticipated environmental perturbation, but they cannot be considered adaptive. Recent work on intelligent control [25] adds an adaptive loop on top of the operational control loop. This is similar to our concept of organiser roles that control the structure of the organization through manipulating contracts. In control-theoretic approaches there is no concept of role or object-role binding. Such systems are not structurally adaptive.

7 Conclusion

Separating management concerns from functional concerns can make systems more adaptive. In this paper we have introduced a framework for creating coordination-

systems that control the interactions between functional roles. These coordination systems can be developed independently and the imposed on functional systems. They are built from a hierarchy of organiser roles that control the contracts between functional roles. ROAD contracts have management and domain function levels. Management contracts specify the type of communication acts and protocols that are permissible between the two parties. Functional contracts specify, among other things, the performance obligations. Abstracting management-contract aspects makes possible, through contract inheritance, the reuse of their communication-control capability in many types of organisational structure.

There are a number of aspects of this framework that need further development. The set of CCAs that defined our example protocol is not complete and somewhat arbitrarily defined. This informality may suffice if operational-management contracts are only application or domain specific. However, if CCAs are to be generalised, a more rigorous approach may be needed. The UML 2.0 Superstructure Specification [26] provides a list of *primitive actions* which may provide the basis for a more formal definition of CCAs. Alternatively, agent communication languages such as [7] may provide the basis of a more rigorous definition. In this paper we have only briefly addressed the nature of organiser roles. While many features of such roles will be domain-specific, there are general principles of organisational viability that need to be elaborated for organiser roles. The discussion of indirect control is also under-developed. In our example, the resource allocation clause gave permission to the Superordinate to access any subtype of Resource. In practice, different roles are likely to have access to different resources. It follows that we need to develop some scheme of resource ownership or access rights. We also assume that the interfaces of the functional roles are compatible if they are to enter into contracts. Issues of functional compatibility and component composition need to be addressed.

References

[1] Bäumer, D., Riehle, D., *Siberski*, W., and Wulf, M. "Role Object" in *Pattern languages of program design 4*, eds. Harrison, Foote, and Rohnert, H. Addison-Wesley, 2000, pp. 15-32.

[2] Kendall, E. A., "Role Modelling for Agents System Analysis, Design and Implementation" *First International Symposium on Agent Systems and Applications IEEE CS Press*, 1999

[3] Kristensen, B. B. and Osterbye, K., "Roles: Conceptual Abstraction Theory & Practical Language Issues" *Special Issue of TAPOS on Subjectivity in Object-Oriented Systems*, 1996

[4] Lee, J. S. and Bae, D. H., "An enhanced role model for alleviating the role-binding anomaly" *Software: practice and experience*, vol.32, 2002, pp. 1317-1344.

[5] Colman, A. and Han, J., "Operational management contracts for adaptive software organisation," *Proc. Australian Software Engineering Conference (ASWEC 2005),* 2005.

[6] Riehle, D. "Bureaucracy" in *Pattern Languages of Program Design 3*, eds. Martin, Riehle, and Buschmann. Reading, Massachusetts: Addison-Wesley, 1998, pp. 163-186.

[7] The Foundation for Physical Intelligent Agents, *FIPA Communicative Act Library Specification* http://www.fipa.org/specs/fipa00037/, 2002, *last accessed 27 Aug 2004*

[8] Bracciali, A., Brogi, A., and Canal, C., "Dynamically Adapting the Behaviour of Software Components," *Proc. Coordination'02 LNCS 2315,* York, UK, 2002.

[9] Han, J. and Ker, K. K., "Ensuring Compatible Interactions within Component-based Software Systems" *Proc.10th Asia-Pacific Software Engineering Conference(APSEC)* 2003.

[10] Meyer, B. *Object-oriented software construction*, New York: Prentice-Hall, 1988.

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

[12] Colman, A. and Han, J., "Implementation of Contracts using Association Aspects", SUT Report, SUTICT-TR2005.04/SUT.CeCSES-TR007 www.it.swin.edu.au/centres/CeCSES, 2005.

[13] Eclipse Foundation, *AspectJ* http://eclipse.org/aspectj/, 2004, *last accessed 7 Oct 2004*

[14] Sullivan, K., Gu, L., and Cai, Y., "Non-modularity in aspect-oriented languages: integration as a crosscutting concern for *AspectJ*," *Proc. of the 1st international conference on Aspect-oriented software development, AOSD 02,* Enschede, The Netherlands, 2002.

[15] Sakurai, K., Masuharat, H., Ubayashi, N., Matsuura, S., and Komiya, S., "Association Aspects," *Proc. of the Aspect-Oriented Software Development '04,* Lancaster U.K, 2004.

[16] Kendall, E. A., "Role model designs and implementations with aspect-oriented programming." *Proc. Object-Oriented Systems, Languages, and Applications,* 1999.

[17] Arbab, F., "What Do You Mean, Coordination? " *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, vol.March, 1998

[18] Andrade, L., Fiadeiro, J. L., Gouveia, J., Koutsoukos, G., Lopes, A., and Wermelinger, M., "Patterns for coordination ," *Coordination '00 LNCS 1906,* pp. 317-322, 2000.

[19] Wermelinger, M., Fiadeiro, J. L., Andrade, L., Koutsoukos, G., and Gouveia, J., "Separation of Core Concerns: Computation, Coordination, and Configuration," *Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, 2001.

[20] Andrade, L., Fiadeiro, J. L., Gouveia, J., and Koutsoukos, G., "Separating computation, coordination aand configuration" *Journal of Software Maintenance and Evolution: Research and Practice* , vol.14(5) , 2002, pp. 353-369 .

[21] Zambonelli, F., Jennings, N. R., and Wooldridge, M., "Developing multiagent systems: The Gaia methodology" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol.12(3) , 2003, pp. 317-370 .

[22] Juan, T., Pearce, A., and Sterling, L., "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*, 2002, pp. 3-10.

[23] Odell, J., Parunak, H. V. D., Brueckner, S., and Sauter, J., "Changing Roles: Dynamic Role Assignment" *Journal of Object Technology, ETH Zurich*, vol.2(5) , 2003, pp. 77-86.

[24] Zambonelli, F., Jennings, N. R., and Wooldridge, M. J., "Organisational Abstractions for the Analysis and Design of Multi-Agent Systems," *Workshop on Agent-oriented Software Engineering ICSE 2000,* 2000.

[25] Herring, S. and Kaplan, C., "Viable Systems: The Control Paradigm for Software Architecture Revisited" *Australian Software Engineering Conference*, 2000, pp. 97-105.

[26] Object Management Group, *UML 2.0 Superstructure (Final Adopted specification)* http://www.uml.org/#UML2.0, 2004, *last accessed 13 Oct 2004*