Operational Management Contracts for Adaptive Software Organisation

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

Abstract

As modern computing environments become more open, distributed and pervasive, the software we build for those dynamic environments will need to become more adaptable and adaptive. We have previously introduced the ROAD framework for creating flexible and adaptive software structures. This framework is built on a distinction between functional and management roles. Management roles participate in contracts that regulate the global-flow of control through a structure of objects and roles. This paper shows how these operational-management contracts can be defined. Such contracts specify the permissible interactions between objects playing functional roles within an organisational structure. Association aspects are shown to have the expressiveness needed to represent such management contracts.

1. Introduction

As modern computing environments become more open, distributed and pervasive, the software we build for those dynamic environments will need to become more adaptable and adaptive. *Organisational* viability of software is required to make software adaptable and adaptive in changing environments [3]. In order to achieve organisational viability we need to represent organisational aspects of software. This paper addresses how software organisation might be represented, at both design and code levels, so that it is amenable to adaptation. This is a prerequisite if we are to create viable adaptive software systems whose organisational representation can be manipulated.

One way to create adaptable software is to create a loosely coupled structure and to create the relationships between the nodes in that structure as late as possible. The relationships in the structure are regulated according to the changing environmental demands. In this approach, software organisation can be viewed as the maintenance of viable arrangements of elements and the regulation of the flow of control in the structure.

This paper extends our work in [3] on the ROAD framework which models software as a decoupled network of roles and objects. The management of the software is seen as a 'separate concern' from the functional aspects of the software. In particular, we show how *management contracts* can be used to regulate the flow of control through a network of roles. The form of such contracts is defined, and we also demonstrate how *association aspects* [18] can be used to implement them.

1.1. Example

Let us consider an example to illustrate how organisational abstractions can be modelled. This example models 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 varied roles, sometimes simultaneously.

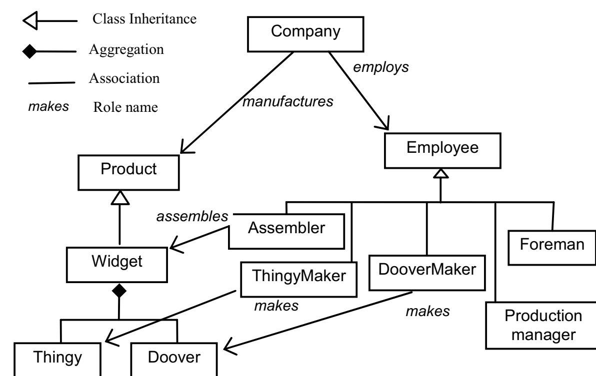

Figure 1. Conventional Object-oriented Class Model

In Figure 1 above, roles are design-level descriptions of association between classes. The associations between classes are fixed at design-time in method invocations and inheritance relationships. The associations between classes/objects cannot be dynamically created or richly described. For example, what *types* of interaction are permissible between an Assembler and a Foreman, and do these interactions differ from those types of interaction between an Assembler and a ThingyMaker? Can an Assembler tell a Foreman what to do by invoking its methods? In object-oriented design, there is no organisational level description in terms of the control of the system. The global flow of control through the structure cannot be represented. Only particular sequences of specific interactions can be shown (e.g. in sequence diagrams). Finally, in conventional object-oriented design, roles are implicit to the objects that play them. There can be no dynamic adaptation of the structure of the relationships between objects and roles in response to changing demands on the system.

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

We will rework the above example to show how management contracts can be defined that regulate the associations between roles in a decoupled structure.

1.2. Structure of this paper

Section 2 gives an overview of the ROAD framework which provides the context for the discussion of role contracts. The model of the network formed by these management contracts is an organisational description of the software system based on the flow of control. Section 3 characterises the different types of control in such a network according to whether it is direct or indirect, and according to the scope of control. In Section 4 we use this characterisation of control to examine in more detail operational-management roles and the contracts by which they are associated. We will define these contracts in terms of *control-communication acts* (CCAs) and demonstrate how to formalise them into contracts. We examine the nature of such contracts then define the expressive requirements needed to represent them. Section 5 provides a brief overview of *association aspects*. We then show how association aspects meet the expressive requirements needed to represent operational-management contracts defined in the previous section. Section 6 examines related work and Section 7 draws conclusions and outlines further work.

2. Overview of the ROAD framework

The Role-Oriented Adaptive Design (ROAD) framework (not to be confused with the Roadmap [7] agent-oriented methodology) is a method for creating adaptable and adaptive object-role software structures. The ROAD framework extends work on role and associative modelling in [1,9-11]. In this section we give a brief contextual overview of our ROAD framework. A more extensive description of the basis of this framework can be found in [3].

A role is an interface of an object that satisfies responsibilities to the system as a whole. We follow Kristensen [10] in viewing roles as separate design and implementation entities. Roles can be added to, and removed from, objects. [10] 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.

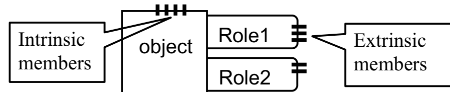

Figure 2. Object and role members

In our view, this 'core functionality' is the situated computational and communication capabilities of the object. Extrinsic members implement the domain function roles of the object.

Returning to our example in the Introduction, rather than modelling a Foreman as a subclass of Employee, Foreman becomes a role an Employee can play. The static inheritance relationships with the Employee class would be removed. These are replaced by potential role-object bindings. Note that Widget would not be treated as a role of Product because in the problem domain Products cannot change roles.

From the basis of decoupled class-role structures, ROAD defines organisational levels of abstraction. The ROAD framework extends previous work on roles by making an explicit distinction between functional and management roles. Three types of role are defined: functional, operational-management and organisational- management roles.

Functional roles are focused on first-order goals — on achieving the desired problem-domain output. Functional roles constitute the process as opposed to the control of the system. Some functional roles are coupled to the environment through system i/o. In Figure 3 below, the Employee e1 playing the role of Foreman can invoke action (e.g. 'make 10 widgets') in the WidgetMaker role played by the e2 Employee object. The discussion of the binding between functional roles and objects is outside the scope of this paper.

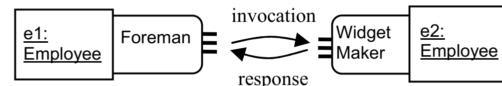

Figure 3. Association between functional roles

Operational-management roles, on the other hand, focus on regulating the relationships between roles. They define contracts between roles based on a separation of management control from process. Operational management roles have no direct connection with the environment. Extending the example from Figure 3 above, we can characterise the *management* relationship between a Foreman and a WidgetMaker as a Supervisor-Subordinate relationship – the Foreman in the operational-management Supervisor role and the WidgetMaker in the operational-management Subordinate role. The relationship between objects, functional roles, operational-management roles, and contracts is illustrated in Figure 4 below.

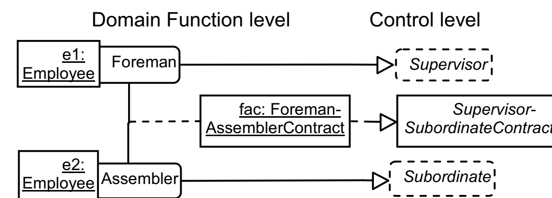

Figure 4 Operational-management roles

1530-0803/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

A network of operational-management roles bound by contracts regulates the flow of control through the system — the *organisation* of the software. In this paper, we are focussing on these *operational-management* roles and contracts.

Organisational-management roles maintain a reflective representation of the system's organisation, and have mechanisms for restructuring the organisation by creating/destroying role-object bindings and dynamic role-role associations. The controllers (objects/agents/ humans) that play organisational management roles are responsible for deciding what objects will play the various functional roles, and for the restructuring of the network of operational-management roles. These controllers are linked to the environment and monitor the performance of the software system in terms of its goals. This forms an *adaptive loop*. The discussion of organisational management roles is beyond the scope of this paper.

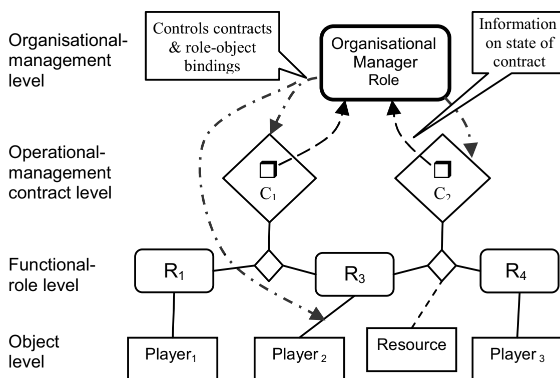

Figure 5. Associations in levels of abstraction in ROAD.

Figure 5 above illustrates the relationships between the different types of roles in the ROAD framework. In summary, the ROAD framework achieves adaptivity through creating decoupled object-role structures. The roles in this initial structure are *functional roles*. A cross-cutting management role-structure is then created from *operational-management* roles. The creation of links between these management roles forms an abstract organisational structure. This structure represents the topology of the organisation and global control flow based on permissible role interactions. The functional roles are bound to the network of operational-management roles. This binding creates a *domain-specific* organisational structure. An *instantiated* organisational structure can then be created by binding functional roles to objects. Such a structure is adaptable. Run-time adaptivity is achieved by defining *organisational-management roles*. These roles control and maintain the organisation by creating and destroying object-role bindings and dynamic role-role associations.

In this paper, we focus on the characterisation of *operational-management roles* and their association contracts within this framework. We show how these roles can be implemented as *aspects* that *cross-cut* functional roles and objects.

3. Control in a management network

We define organisation as the *global flow of control* through a system. Unlike natural systems, in which organisation is emergent, software systems are designed to achieve goals. A shortcoming of many supposed organisational descriptions is that they reduce the description of organisation to just the topological structure. In our definition, organisational descriptions of designed systems are means-end functional descriptions. A complete organisational description would need to indicate how goals are transmitted through the system, how the system changes in response to changing goals and environmental perturbations, and how the system maintains its organisational viability. Such organisational descriptions can be based on the conceptual separation of control from process — that is, the separation of management of the process from the process itself. An organisational/managerial aspect of a structure facilitates the intentional flow of control through a structure of management roles, whereas relations between functional roles define the dataflow through the structure. Management functions can be, to some extent, characterized in a domain-independent way. These functions include coordination, goal-transmission, regulation, accounting, resource-allocation, auditing and reporting. The organisational perspective is one of a number of possible perspectives, but it is a perspective that allows us to explicitly represent and incorporate adaptive mechanisms into a system.

Control in an organisation can be characterised as direct or indirect. *Direct* control is concerned with positive goal propagation through the structure. In a hierarchical structure, direct control is a chain of command. Each node in the structure reinterprets the goal(s) passed down to it. The node then operationalises the goal(s) either by executing a process itself or setting goals for other roles. *Indirect* control is control through constraint – for example, the regulation of a process though the allocation of resources to roles performing the process. Resources can be thought of as objects that do not perform management functions within the organisational structure.

Either individual components, or parameters that are system/subsystem wide, can be controlled. The table below illustrates various mechanisms for control categorised by *scope* of control and *type* of control.

IEEE COMPUTER SOCIETY

Table 1. Control type versus scope of control and example mechanisms of control

Control type \ Scope	Individual component	(Sub)system
Direct control	Command / goal setting	Prescriptive rules
Indirect control	Resource allocation Individual policies	Proscriptive rules Group policies Norms

In many systems, a combination of these modes of organisation will be present. In this paper, we focus on direct and indirect control of individual components (the shaded area in the above table) through the assignment of organisational responsibilities and the creating of structure for management control.

In the next section we characterise such operational-management associations and contracts in more detail.

4. Operational-management role associations and contracts

The separation of operational-management roles from functional roles gives us a way to describe the organisational topology of the system and the control regime of that structure. Management hierarchies of any complexity, such as that shown in Figure 6 below, can be created with such operational-management roles. The static representation of such a hierarchy would have the appearance of a business's organisational chart.

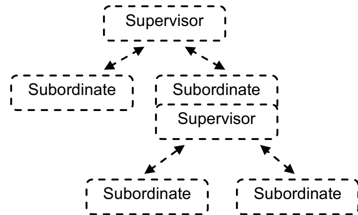

Figure 6. Abstract organisation structure of operational-management roles

A domain-specific organisation is created when functional-roles are bound to operational-management roles. In Figure 7 below, an organisational structure has been created for our Widget department using Supervisor-Subordinate operational-management role contracts. In order to simplify the diagram, functional contracts have been drawn as diamonds. Every functional role in the organisation has a position in the control structure and thus has one or more associated operational-management roles — one for every type of role-role association in which the functional role participates. In other words, there is a correspondence between functional role-role associations and operational-management role-role associations. We call this binding *association inheritance.* Note that the structure is still abstract because no objects have yet been assigned to roles.

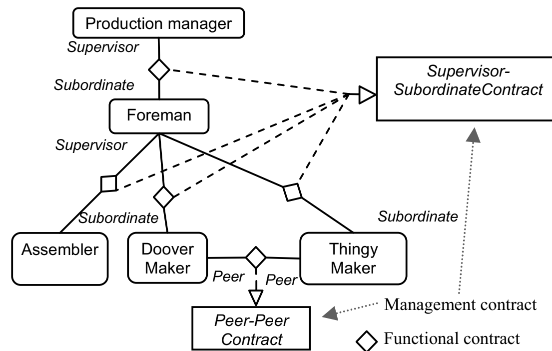

Figure 7. Domain specific abstract organisational structure

Operational-management *contracts* restrict the interactions between objects playing particular roles. In object-oriented languages such as Java, a target object will respond to any valid invocations of its public methods from objects that have the target in their scope. The scoping of accessibility of such methods can only be structured in a primitive way using accessibility modifiers, program blocks, packages, namespaces etc. Operational-management contracts *restrict* the type of method that one role can invoke in another role, or restrict what methods it will respond to from another role. From our example above, the Supervisor-Subordinate contract restricts interactions between the objects playing the WidgetMaker and the Foreman to certain *types* of interaction. For example, a WidgetMaker cannot tell its Foreman Supervisor what to do. These contracts also restrict interaction between particular instances of object playing the roles. For example, the method `WidgetMaker.setProductionTarget()` can only be invoked by the WidgetMaker's own Foreman.

Supervisor-subordinate management associations are only one type of operational-management contract. Others could include:

- Auditor-auditee
- Peer-peer
- Supply-chain predecessor-successor
- Production-line predecessor-successor

4.1. Control-Communication Acts

We characterise the types of operational-management contracts in terms of the control communication between roles that are party to the contracts. Such control communication can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the control aspects of the communication from the functional aspects. We can define a simple set of CCAs in terms of the direct/indirect control distinction made in the previous section. *Direct control* is direct invocation of

IEEE COMPUTER SOCIETY

action in another role (command) or the setting of a goal state in another role. *Indirect control* is achieved through the allocation or restriction of access to resources. As such, indirect control is inherently referential as it involves three entities – the role granting access to the resource, the role consuming the resource, and the resource itself. We assume resources are passive objects – that is, they do not know which roles are able to access them. Indirect control information therefore passes between the controller and the consumer of the resource, rather than to the resource itself. If access to the resource were to be set by the resource itself, it would need to be represented by a node in the operational-management role network.

In addition to direct and indirect control, *control information* needs to be passed between management nodes. This includes responses to commands or requests – such as *accept*, *refuse* etc. It could also include other information relevant to the regulation of the system - e.g. *busy*, *off-line* etc.

Table 2. Example of Control-Communication Act Primitives

Type of communication	Communicative control acts
Direct Control	DO, SET_GOAL
Indirect Control	RESOURCE_ALLOC(r), RESOURCE_REQUEST(r)
Information	ACCEPT, REFUSE, INFORM, QUERY

As an example, Table 2 above defines a set of primitives suitable to a hierarchical organisation. Note that because indirect CCAs express a ternary relationship, they carry a reference to a resource `r`. The above set 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.

4.2. Operational-management contracts

The concept of a contract is commonly used in software engineering. For example, design-by-contract [12,13] 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 in that they only explicitly express the conditions for one party – the other party (client) is anonymous. In the real world however, contracts always have at least two parties. They are a type of association that expresses the obligations and responsibilities of parties to each other. Contracts can be unique (e.g. a contract to build an opera house) or follow a standardized type (e.g. contract for sale for a residence).

Contracts can have a number of incarnations: *form* (*à la* class), *instantiation* (*à la* object) and *execution*. The *form* (type) of a contract sets out the mutual obligations and interactions between parties of a particular class (e.g. vendor and purchaser). A contract is *instantiated* with an identity when values are put against the variables in the contract *schedule* (e.g. vendor and purchaser are named, date of commencement agreed etc.) and the contract is signed. Contracts can also be thought of as having an *execution state* in terms of the fulfilment of the various clauses of the contract.

Operational-management contracts are examples of such associational contracts – they define the form of an ongoing control association between two roles in an abstract organisational structure. The *form* of such contracts contains:

1. Variables defining the parties to the contract. These variables are of a particular type of participant that can enter into the contract.
2. A protocol that defines the allowable *types* of interaction between those parties. In operational-management contracts, these protocols are described in terms of the control relationships between the parties (e.g. A has the power to tell B what to do) rather than the functional relationships (A tells B to do a particular action). In terms of software, the protocols define the allowable types of method invocation that one type of operational-management role (e.g. supervisor) can make on another (e.g. subordinate) and the expected response.
3. Other clauses in the form of contract define variables that relate to the execution of the contract. These include conditions relating to commencement, continuation, performance and termination of the contract. Operational-management contracts are on-going in that they define the control relationships between the parties whilst there is an organisational relationship between the parties. In this sense they are more like a service-level-agreement or an employment-contract, than they are like a contract of sale. In a commercial contract such variables are part of the contract *schedule*.

An *instance* of an operational-management contract is created when the variables in the contract schedule are given values — in particular when operational-management roles are bound to functional roles (e.g. a Supervisor role is bound to a Foreman role). The rules for communication that are defined in the contract protocol are mapped to the method invocations in the functional roles. Performance criteria can also be attached to the execution of various clauses in the contract or to the contract as a whole.

Information on contract *execution* needs to be stored, along with the static information described above. This dynamic information is needed to ensure that the terms of the contract are being met, and includes information on the state of the relationship between the parties (e.g. active, inactive, suspended, in-breach, terminated etc.), and the state of any

interaction defined by the protocols (e.g. A has sent a Query to B and is waiting for a response). In the real world, commercial contracts are just passive artefacts so this information is maintained by the parties themselves (or their agents). In software contracts, it makes sense to store this dynamic information in the contract itself.

Using the primitives we defined in the previous section, a Supervisor-Subordinate contract could be defined as in Table 3 below. When a functional role in an organisational structure is bound to an operational-management role using such a contact, all functional role invocations and responses are associated with CCA primitives.

Table 3. Example form of operational-management contract

Operational-management Contract	
Name	Supervisor-Subordinate
Party A	Supervisor
Party B	Subordinate
A initiated	DO → ACCEPT, INFORM SET_GOAL → ACCEPT, INFORM INFORM → — QUERY → INFORM RES_ALLOC → ACCEPT
B initiated	INFORM → — QUERY → INFORM or REFUSE RES_REQ → RES_ALLOC or REFUSE

In summary, the expressive requirements needed to represent operational-management contracts in terms of state and behaviour are:

State: Each instance of a contract must include:

- the name of the parties (i.e. the functional roles or objects playing those roles)
- a FSM of the state of communication between the parties as defined by the protocol. It is this representation of the state of the association (viable or otherwise) that allows the organisational manager to maintain a representation of the state of control-flow through the organisation.

Behaviour represented in the contract includes:

- a protocol definition of permissible types of interaction between the parties
- a mapping from the protocol to the interface signatures of the respective functional-roles. This requires adherence to a naming standard for the methods so that they can be associated with types of invocation
- a mechanism for enforcing the protocol on communications between functional-roles

In the next section, we show how association-aspects [18] can be used to implement operational-management contracts with the above expressive requirements.

5. Using Association-Aspects to implement operational-management contracts

This section shows how we can implement operational-management contracts using the *association aspect* extension [18] to AspectJ [4]. We begin with a brief discussion of AspectJ aspects and association aspects and how they can be used to model behaviour between groups of objects. The subsequent subsections outline the steps that are taken to create a contract:

1. The elements of a contract related to *direction* and *restriction* on communication are defined using *pointcuts*.
2. Contract *clauses* are then created from these elements.
3. Actions to be taken when a contract clause is triggered are then defined using aspect *advice*.
4. A contract is constructed from its clauses and other elements in its schedule.
5. An instance(s) of a contract is created.

Aspect-oriented methods and languages seek to maintain the modularity of separate cross-cutting concerns in the design and source-code structures. As pointed out above, the organisation of software as expressed in a network of *operational-management* roles, cross-cuts the program structure defined by the *functional-roles* and classes.

The AspectJ 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.

While AspectJ-like aspects have previously been used to add role behaviour to a single object [9], as far as we are aware they have not been used to implement associations between roles. Aspects as currently implemented in AspectJ do not easily represent the behavioural associations between objects [19]. 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 groups of objects.

Sakurai et al. [18] propose the use of *association-aspects* to allow an aspect instance to be associated with a group of objects. Such association-aspects meet the expressive requirements that we defined in the previous section. *Association-aspects* are implemented with a modification to the AspectJ compiler to handle

IEEE COMPUTER SOCIETY

an additional pointcut primitive. Association-aspects allow aspect instances to be created in the form

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

where a1 is an aspect instance and o1 and oN are a tuple of two or more objects associated with that instance. Association-aspects are declared with a `perobjects` modifier that takes as an argument a tuple of the associated objects.

```
aspect AnAssociationAspect perobjects(o1, o2){
   //aspect variables
   //pointcut declarations
   //advice methods
}
```

Figure 8 below is a schema that sets out the relationship between the code-level constructs (such as join points, named pointcuts, advice), and the functional-role and operational-management roles. The contract, as implemented by the association aspect, defines pointcuts that match particular types of communication between particular parties. Actions from the contract (in the form of advice) are woven into the code of the functional role. If a control communication triggers a clause in the contract the respective action is executed.

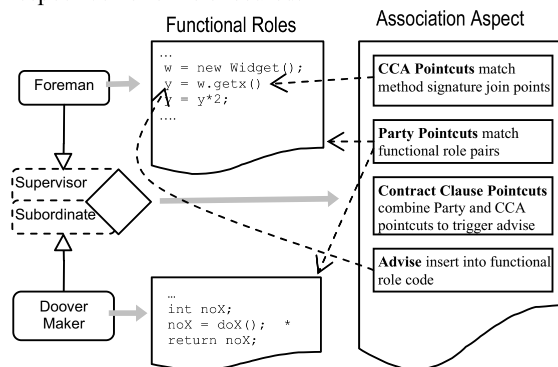

Figure 8. Using association aspects to implement contracts

The following subsections will explain this schema in more detail.

5.1. Defining the contract elements

To create an operational-management contract type, as defined in Section 4, we need to define the parties that participate in the contract, and which control-communication acts (CCAs) each of the parties can use. To do this we create three types of pointcuts:

1. Party pointcuts that match the parties to the contract.
2. CCA pointcuts that define the types of message.
3. Instances of these two above basic types of pointcut are then composed into pointcuts that represent particular clauses in the management contract. These composite pointcuts define *who* can say *what*.

Party pointcuts represent the parties to the contract and the direction of communication between those parties. For example, in a contract between two operational-management roles (of type MRole) there would be two party pointcuts: a `aToB` pointcut that represents communication from party A to party B, and a `bToA` pointcut that represents communication the other way. The definition in AspectJ is as follows:

```
pointcut aToB(MRole a, Mrole b):
    associated(a,b) && this(a) && target(b));
```

The `associated(a,b)` condition is an AspectJ extension from [18]. In this case it ensures that the parties, represented by the particular MRole variables `a` and `b`, are associated in a contract. The `this(a)` condition ensures `a` is making the call. The `target(b)` condition ensures that `b` is the target of the communication.

CCA pointcuts use a mixture of primitive pointcuts provided by AspectJ and pattern matching on the method signatures to enforce the communication protocol between the functional roles. If the CCA types cannot be distinguished by primitive pointcuts alone, a naming-convention is required that identifies the method signature with particular CCAs in the contract. To achieve this in our example we define the convention that: 'an abbreviation of the CCA prefixes the method'. For example the name `setG_Daily-WidgetQuota()` enables a mapping to be created between the functional method that sets the daily quota of Widgets, and the SetGoal CCA primitive defined in the operational-management contract. Where CCAs are referential, as is the case with resource allocation, the method signature is distinguished by the type of parameter. In our example, all resources implement a `Resource` interface. The CCA pointcut ResourceAllocate could be defined as follows:

```
pointcut resAlloc() : call(* ra_*(Resource));
```

This pointcut called `resAlloc` matches any method *call* that

- begins with the characters "ra_"
- returns any type
- has a variable of type Resource as a parameter.

5.2. Defining the clauses of the contract

Contract clause pointcuts are the combination of a Party pointcut and a CCA pointcut. For example the pointcut below is Clause 1 (`a1`) of the contract. It says that Supervisor **`sup`** has the authority to allocate resources to the Subordinate **`sub`**.

```
pointcut a1(Supervisor sup, Subordinate sub)
: aToB(sup, sub) && resAlloc();
```

A clause is defined for every CCA that can be initiated by either party. In the case of the SupervisorSubordinate contract as defined in Table 3 above, there are eight clauses required in all – five

IEEE COMPUTER SOCIETY

governing communication from the Supervisor to the Subordinate (a1..a5), and three governing communication from the Subordinate to the Supervisor (b1..b5). We can define further clauses `a0` and `b0` that is the compound of all the `aX` and `bX` clauses. For example `a0` would be defined as:

```
pointcut a0(Supervisor sup, Subordinate sub):
    a1(sup,sub)|| a2(sup,sub)|| a3(sup,sub)||
    a4(sup,sub)|| a5(sup,sub);
```

5.3. Defining the effect of the contract clauses

Once the clauses of the operational-management contract have been defined, the actions that occur when particular clauses of the contract are triggered need to be defined. These actions take the form of aspect advices. For communications between functional roles that are in accord with the clauses of the contract, no modification to the communication is necessary. However, the state machine that keeps track of the communication within the association contract is updated, both when the method call is made (Party A has made a request under the terms of the contract) and when the method returns (Party B has responded in appropriate form). Updating of the contract state can be done with `before()` and `after()` advices either for individual contract clauses (e.g. `a1`) or a compound clause (`a0`) :

```
before() : a1{...}/* update contract state
     machine and add any extra management
     functions e.g.accounting */
after() returning : a1{... } //ditto
```

All communication that does not conform to a contract clause should be prohibited. This is done with `around()` advice or having a `before()` advice throw an exception. Around advices prevent the execution of the invoked code. Any method call between parties to the contract that does not correspond to terms of the contract (e.g. `!a0` as defined above), throws an error.

```
before(...): !a0{
   throw new InvalidCCA(thisJoinPoint);}
```

5.4. Putting the contract together

The Party and CCA basic pointcuts are the common basis for all two-party operational-management contracts. Given, this we can define an abstract `ManagementContract (MContract)` aspect that contains these basic pointcuts, rather than having to define each management contract from scratch. Likewise, all operational-management roles, such as Supervisor and Subordinate, implement the `ManagementRole (MRole)` interface. Figure 9 below shows these relationships.

The code fragments below come from a program written to test the implementation of contracts using association aspects. Notated code for this program can be found at [2].

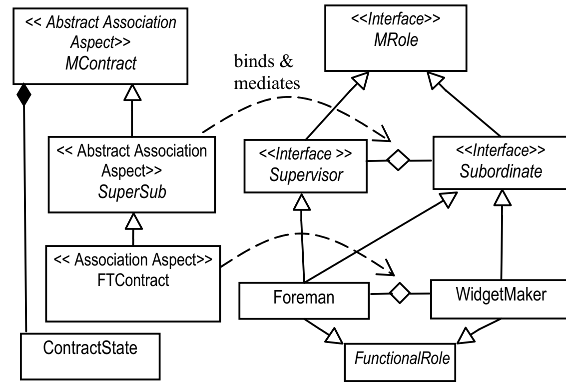

Figure 9. Inheritance Diagram of Aspects and Roles

The definition of a basic abstract two-party contract is as follows:

```
public abstract aspect MContract {
   ...
   protected ContractState cs; /* could be
       overridden by sub-aspect */
 ...
   //Party communication direction pointcuts
   abstract pointcut aToB (MRole a, MRole b);
   abstract pointcut bToA (MRole a, MRole b);
             //CCA pointcuts
   pointcut doIt() : call(* do_*(*));
   pointcut setGoal() : set(void setG_*(*));
   pointcut inform() : set(void inf_*(*));
   pointcut query() : call(* qry_*(*));
   pointcut resAlloc(): call(* ra_*(Resource));
       //returns reference to resource
   pointcut resReq():call(Resource rr_*(String));
       //parameter name of requested resource
       //returns reference to resource or null}
```

The form of the Supervisor-Subordinate contract aspect can now inherit from the general management contract as below. The contract from Table 3 above is replicated with numbered clauses (a1..a5, b1..b3).

Direction	Clause	CCA
Supervisor initiated	a1	DO → ACCEPT, INFORM
	a2	SET_GOAL → ACCEPT, INFORM
	a3	INFORM → —
	a4	QUERY → INFORM
	a5	RES_ALLOC → ACCEPT
Subordinate initiated	b1	INFORM → —
	b2	QUERY → INFORM, REFUSE
	b3	RES_REQ → RES_ALLOC, REFUSE

```
public abstract aspect SuperSub extends
     MContract{
   ...
   /*define contract clauses from directional
   Party and CCA pointcuts defined in the
   abstract ManagementContract parent class */
   pointcut a1(Supervisor sup, Subordinate sub)
       : aToB(sup, sub) && doIt();
   pointcut a2(Supervisor sup, Subordinate sub)
       : aToB(sup, sub) && setGoal();
   ...
   pointcut b1(Supervisor sup, Subordinate sub)
       : bToA(sup, sub) && inform();
   ...
   //all valid clauses
```


IEEE COMPUTER SOCIETY

```
  pointcut a0(Supervisor sup,Subordinate sub):
      a1(sup,sub)||a2(sup,sub)||a3(sup,sub)||
      a4(sup,sub)||a5(sup,sub);
  pointcut b0(Supervisor sup,Subordinate sub):
      b1(sup,sub)||b2(sup,sub)||b3(sup,sub);
  pointcut c0(Supervisor sup,Subordinate sub):
      a0(sup, sub) || b0(sup, sub);

/*advices that define the actions when a
contract clause is triggered*/
   before() : c0{
   cs.update(thisJoinPoint);}
   /* update contract state machine. Add any extra
     management functions e.g.accounting*/
   after() returning : c0{
      cs.update(thisJoinPoint);}
   before(): !c0{
      throw new InvalidCAA(thisJoinPoint);}
}
```

We can now define a concrete aspect based on the abstract Supervisor-Subordinate contract. Below is a functional contract that defines the association between a Foreman and a ThingyMaker. In addition to monitoring and controlling the CCAs between parties to the contract, functional contracts can define performance requirements specific to the parties. In real-world terms, functional contracts fill in the details of the contract schedule attached to the *form* of contract defined by the operational-management contract. For example, a functional contract may require a ThingyMaker to make a Thingy with x seconds.

```
public aspect FTContract extends SuperSub
     perobjects(Supervisor, Subordinate) {
  ...
  public FTContract(Foreman f, ThingyMaker t){
   ...
   associate(f, t);//creates association
  }
  //instantiate directional pointcuts
  pointcut aToB(Supervisor f, Subordinate t):
     this(f) && target(t) && associated(f, t);
  pointcut bToA(Supervisor f, Subordinate t):
     this(t) && target(f) && associated(f, t);
 ...
//define methods for revoking and reassigning
contract
//define functional performance pointcuts}
```

5.5. Creating the contract instance between functional roles

We now apply these programming constructs to operational-management contracts using the Supervisor-Subordinate contract as an example. The creation of an instance is done as follows:

```
class Foreman implements Supervisor,
                                   Subordinate {… }
class WidgetMaker implements Subordinate {… }
//create instances of functional-roles
Foreman f = new Foreman();
WidgetMaker w = new WidgetMaker();
ThingyMaker t = new ThingyMaker ();
/* create the Foreman- ThingyMaker
(FTContract) contract instance that binds the
functional-roles also passes reference to
organisational-manager creator */
FTContract ft1 = new FTContract(this, f, t);
```

Communications between the Foreman and the ThingyMaker now conform to the SuperSub contract.

6. Related work

Our approach extends work on role and associative modelling in [1] where roles are first-class design and implementation entities [1,6,8,10,11]. Kendall [9] 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 an organisational level of abstraction by defining management roles. They are concerned with role-object bindings rather than role contracts.

The approach here is similar to OOram [17] to the extent that roles are nodes in an interaction structure (role-model). In [17] role-models can be based on any suitable separation of concerns, whereas here we distinguish domain and abstract management concerns, and we represent collaborations with separate contract entities. Responsibility-driven design (RDD) also focuses on collaborations between roles, but contracts apply to individual objects and are seen as “really meaningful only in the programmer's mind” [20]. Operational-management roles in ROAD can be viewed as a type of RDD object-role stereotype. Such stereotypes may provide the basis for defining an expanded set of operational-management contracts.

[16] and [18] propose different solutions to modelling the behaviour between groups of objects. The former’s AOP language Eos aspects can be created to represent behavioural relationships, however it selects advice execution associated with a target object. Sakurai [18], on the other hand, modifies the AspectJ compiler to handle the additional *associated* pointcut primitive. We have used the latter approach because it allows selection of the aspect instance based on any of the objects in the association.

The notion of CCA in this paper is derived from the concept of a *communication act* in multi-agent systems (MAS) agent communication languages such FIPA-ACL [5]. CCAs, as defined here, are more restricted in their extent. CCAs deal only with control communication, and do not have to take intentionality of the other parties into account. Work on roles has also been undertaken in MAS [7,15,21]. In particular, [22] 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.

IEEE COMPUTER SOCIETY

7. Conclusion and further work

In this paper we have shown how association aspects can be used to implement operational-management contracts. These contracts create relationships between functional roles and regulate the flow of control communication between them. In the ROAD framework, such contracts form a flexible organisational network that is designed to make software structures more adaptive to changing environments and goals.

Here we will limit the comments on further work to the area of operational-management contracts. There are a number of open issues. This paper has used the supervisor-subordinate association as an example. Contract protocols need to be developed for other operational-management associations, such as those listed in Section 4. 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 [14] provides a list of *primitive actions* which may provide the basis for a more formal definition of CCAs. Alternatively, agent communication languages such as [5] may provide the basis of a more rigorous definition.

The discussion in this paper has been limited to two-party contracts. Examples of protocols that involve more than two parties need to be developed. Examples of protocol sequences (e.g. negotiation protocols) that are more than just a single invocation-reply pair, also need to be 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 .

There are unresolved questions relating to the execution phase of contracts. Are the contracts managed externally by observing the state of the contract, or is breach/failure in the contract report triggered by the contract itself? For example, in the case of a subordinate's failure to meet the terms of a contract, presumably a supervisor would try to find another subordinate to perform the task. If that is not possible, the supervisor would report to the next higher level. Such issues will need to be resolved in the context of the ROAD framework.

8. References

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

[2] Colman, A., *Notated Java code for the Implementation of Contracts using Association Aspects* ,ver1.0, www.it.swin.edu.au/personal/acolman/, 2004.

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

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

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

[6] Fowler, M., "Dealing with Roles," *Proc 4th Annual Conference on the Pattern Languages of Programs,* Monticello, Illinois, USA, 1997.

[7] Juan, T., Pearce, A., and Sterling, L., "ROADMAP: extending the Gaia methodology for complex open systems" *Proc 1st Inter. Conf. Autonomous agents and multiagent systems, Bologna, Italy, ACM*, 2002, pp. 3-10.

[8] Kendall, E. A., "Role model designs and implementations with aspect-oriented programming." *Proc ACM Conference on Object-Oriented Systems, Languages, and Applications, Denver, CO*, 1999, pp. 353-369.

[9] Kendall, E. A., "Role Modelling for Agents System Analysis, Design and Implementation" *1st Inter Symposium on Agent Systems and Applications IEEE CS Press*, 1999

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

[11] 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.

[12] McKim, J. and Mitchell, R. *Design by Contract by Example*, Addison Wesley, 2002.

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

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

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

[16] Rajan, H. and Sullivan, K., "Eos:instance-level aspects for integrated system design" *ACM SIGSOFT Software Engineering Notes* , vol.28 (5) , 2003, pp. 297-306.

[17] Reenskaug, T. *Working with Objects : the OOram Software Engineering Method*, Manning Pub. Co., 1996.

[18] 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.

[19] Sullivan, K., Gu, L., and Cai, Y., "Non-modularity in aspect-oriented languages: integration as a crosscutting concern for *AspectJ*," *Proc 1st Inter. Conf. on Aspect-oriented software development, AOSD 02,* 2002.

[20] Wirfs-Brock, R. and McKean, A. *Object Design: Roles, Responsibilities,& Collaborations*, Addison Wesley, 2002.

[21] 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.

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

1530-0803/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY