A Policy Based Role Object Model

Emil Lupu Morris Sloman

Imperial College, Department of Computing,
180 Queen's Gate, London SW7 2BZ, U.K.
E-mail: {e.c.lupu, m.sloman}@doc.ic.ac.uk

Abstract

Enterprise roles define the duties and responsibilities of the individuals which are assigned to them. This paper introduces a framework for the management of large distributed systems which makes use of the concepts developed in role theory. Our concept of a role groups the specifications of management policies which define the rights and duties corresponding to that role. Individuals may then be assigned to or withdrawn from a role, to enable rapid and flexible organisational change, without altering the specification of the policies. We extend this role concept to include relationships as means of specifying required interactions, duties and rights between related roles. Organisations may contain large numbers of similar roles with multiple relationships between them, so there is a need for reuse of specifications. Role and relationship classes permit multiple instantiation and inheritance is used for incremental extension of the organisational structure with minimal specification effort. We also briefly examine consistency and auditing issues related to this role framework.

1. Introduction

Many organisations have handbooks of policies and procedures relating to security or specifications of duties related to positions in the organisation (job descriptions). These policies are usually specified in natural language in terms of groups of employees and organisational roles rather than individuals but are not analysable or directly implementable. We have developed a role framework which can be used to "formally" specify Enterprise viewpoint role concepts, analyse these specifications for consistency and translate them into automated agents for managing distributed systems. We have used the concepts developed in [1],[2] in order to group the specifications of the rights and duties in the organisational structure into roles. We model *rights* as **authorisation policies** which specify what activities a subject[1] is permitted (or forbidden) to perform on a set of target objects. *Duties* are modelled as **obligation policies** which specify what activities a subject must or must not perform on a set of target objects. Users may be assigned to or withdrawn from a role, to enable rapid and flexible organisational change, without changing the policies. We also consider the relationships between multiple roles to define the rights, duties and protocols pertaining to interactions between roles (e.g. right for a manager to assign a task to an assistant, or a protocol by which a general practitioner refers a patient to a specialist). Roles and relationships are seen as the building blocks of the organisational structure. An organisation will have many similar roles, for example a hospital will have several doctors each with their own specific patients. *Role* and *relationship classes* are therefore defined in order to permit multiple instances to be created from a single specification. This requires the definition of policy templates which have to be instantiated with specific targets for every role instance. There is also a need for derivation of specialised roles such as a surgeon from more generic ones such as a doctor, which can be supported by class inheritance.

Multiple policies apply to objects in the system so it is necessary to analyse the policies to detect and resolve conflicts such as a role with a duty to perform an action which is forbidden, or to detect violations of constraints expressed as meta-policies. We briefly examine the conflicts which can occur between management policies, but further information on conflict detection and resolution is given in [3]. We also specify concurrency

[1] We use the term "subject" to refer to an object representing a user, human manager or an automated agent which can initiate activities within the system

constraints relating to ordering of activities regarding role interactions.

There is a growing interest for formalising and automating the clinical process model [4],[5] which has a high degree of inter-personnel collaboration and standardised procedures. Thus, most of the examples used in this paper relate to the responsibilities and interactions in the health care process.

Section 2 introduces the concepts of domains, policies and Meta-policies. In Section 3 we define the roles and relationships. Policy templates, roles and relationship classes are then discussed in Section 4 and consistency issues are presented in Section 5.

2. Management services and policies

2.1 Domains

Large distributed systems may contain millions of objects so it is impractical to specify policies for individual objects. Objects are therefore grouped in domains to specify a common management policy or to structure and partition management responsibility. A **domain** is a collection of objects (actually references to object interfaces) which have been explicitly grouped together for the purposes of management (cf. file system directories or folders). A domain is an object, so may also be a member of another domain. A **domain service** is provided for the manipulation of the membership information. Further, **domain scope expressions** can be specified determining the set of objects to which a policy applies. For example $@D_1$-$@D_2$-O_3 represents the objects that are members of D1 with members of D2 and object O_3 excluded. Our concept of a domain is very similar to that of a directory in a typical hierarchical file system. The policy which applies to a domain will, by default, propagate to sub-domains and to the objects within them, although this propagation can optionally be disabled. A **User Representation Domain** (URD) is a persistent representation of the human within the computing system. When a person logs in, an adapter object (cf. login shell) is created within the URD to act as the interface process between the person and the computer system. Other agents representing the human could also be created in the URD. Details on the domain structure and the relevant services can be found in [6],[7].

2.2 Policies

Management policies are used in order to separate the specification of the behaviour from the software components of the system. They can be dynamically distributed or retracted from the managers, thus changing the behaviour and strategy of the management system without interrupting their functioning. A policy establishes a relationship, between manager and managed object domains, which can be either an **Obligation** specifying what actions subjects must or must not perform on target objects or an **Authorisation** specifying what actions subjects are authorised or forbidden to invoke on target objects. The general format of the policies is given below with optional arguments within brackets:

identifier mode [trigger] subject '{' action '}' target [constraint] [exception] [parent] [child] [xref] ';'

The **mode** of the policy distinguishes between positive authorisation (permitted: A+), negative authorisation (forbidden: A-), positive obligation (must: O+) and negative obligation (must not: O-). The subject represents the set of managers assigned to carry out the actions on the set of target objects. Both sets are specified using domain scope expressions. Positive obligation policies can be triggered by time or by composite events detected within the monitoring system [8]. Constraints limit the applicability of the policy e.g. between the hours of 09.00 and 17.00. The policy format and use is further described in [9]. Examples of policies are:

/ every day nurses are obliged to generate a status log of the drugs used */*

O+ every [1*day] n:@/nurses { generate_log(n) } @/drugs_db;

/ nurses are not authorised to validate discharges */*

A- @/nurses { validate } @/patients/discharges;

Policies can specify actions at different levels of abstraction. A refinement hierarchy can therefore be built from the more abstract policies, which can only be interpreted by humans, to the enactable leaf level policies or rules which can be interpreted by automated components. Tools for policy editing and services have been implemented and are described in [9],[10]. Authorisation policies are translated into access control lists which are interpreted by security agents in the target system [11] and obligation policies are disseminated to distributed automated management agents for interpretation [10].

2.3 Meta-Policies

Several policies may apply to the same objects either to reflect different management functionalities (configuration, performance, monitoring) or because objects may be members of several domains. Conflicts

may then arise between the various policies. We distinguish between modality conflicts which arise from inconsistent modes of the policies, e.g. **O+** and **O-,** and application-specific conflicts such as separation of duties or conflicts for resources [12],[3]. It is necessary to specify constraints pertaining to the attributes of policies in order to avoid application-specific conflicts. We term these constraints meta-policies (policies about permitted policies). They can be expressed as logical predicates applying to sets of policy objects within a domain. For example a conflict of duties stating that the same manager cannot both authorise payment and sign the payment cheque can be written as:

$$\forall P1, P2 \in < \text{domain\_scope\_exp} >$$
$$\text{fail} \leftarrow \text{intersect}(P1.\text{subjects}, P2.\text{subjects}) \wedge$$
$$\text{belongs}(\text{'payment'}, \text{intersect}(P1.\text{targets}, P2.\text{targets})) \wedge$$
$$\text{belongs}(\text{'authorise'}, P1.\text{actions}) \wedge \text{belongs}(\text{'sign'}, P2.\text{actions})$$

Meta-policies, constraining the permitted policies, can be included in a role or a relationship. Other constraints, within a policy expression, limit the applicability of the policy e.g. to a particular time interval, but are different from the meta-policies which express constraints of compatibility between policies.

In the following sections we describe the roles, relationships and associated classes. Consistency problems and modality conflicts will be examined in Section 5.

3. Roles and Relationships

The organisational structure can be represented as inter-related roles. For example a hospital may contain administrative staff, registrars, surgeons, pharmacists, etc. who co-operate in order to provide care to a patient. We initially define a **role** as a group of policies, specifying its obligations and authorisations and a **relationship** as a group of interacting roles. Relationships also contain the policies which specify the duties and authorisations of a role with regards to the other roles e.g. right to assign a task, obligation to provide some information. These policies are considered part of the relationship rather than the individual roles, because their lifetime is dependent on the lifetime of the relationship. We first examine the role as a group of policies then we will show how the role can be extended with constraints and relationships which specify the role interactions.

3.1 Roles as groups of policies

A role groups the policies specifying the duties and rights of a particular position inside the organisation. These policies reference a common subject domain called the **Manager Position Domain (MPD)**. A user is assigned to a role by authorising the user to connect to a proxy object in the MPD which inherits all the rights pertaining to the role and acts as the user's representative in that role (Figure 1). The user interacts with the system via an adapter object in the URD which is similar to an X server in that it provides a separate window for each role. The authorisation policies of the role indicate the permitted actions and can be used to customise the menus or choice of commands presented to the user in the window. This permits a clear separation of activity context for each role to which a user is assigned, and makes sure a user does not use the rights pertaining to one role to perform operations within another role. By analysing the policies referencing the URD it is possible to determine to which roles the user has been assigned. The main advantage of specifying policies in terms of roles rather than individuals is that organisational changes, when individuals are assigned to new roles, does not require any changes to the policy specification relating to the roles. Individual users can still have other policies relating to their URD which have nothing to do with their role, for example private policies permitting access to personal files or global policies about choosing passwords relating to all members of the organisation.

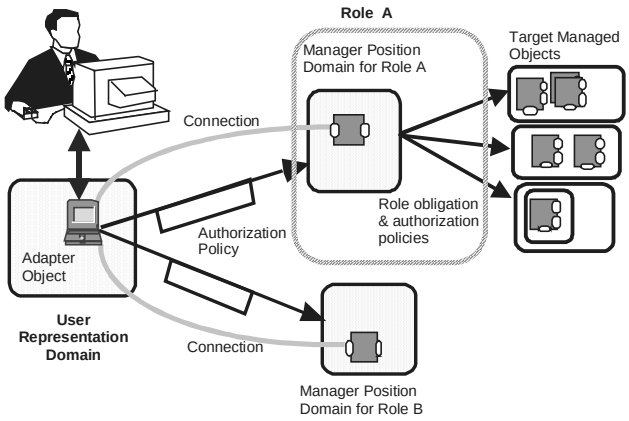

Figure 1 Management roles

The implementation of a role is an object which maintains a reference to the MPD of the role and a table of all the policies which are part of the role. Each entry in this table is composed of a name, unique within the role, and a reference to the policy object which can be resolved by the underlying support system (Figure 2). Note that domains, policies, roles and relationships are objects of the system and therefore can be target objects for other management policies. The management of tasks and of the organisational structure can therefore be specified in terms of the policies and roles themselves. However, the

model described so far is incomplete since it does not cater for relationships between roles and does not provide any means of specifying consistency or synchronisation constraints relating to the activities of the roles.

3.2 Extended Roles

Obligation policies may synchronise only on the events which trigger them. This is a low-level mechanism which requires an administrator specifying policies and roles, to know which events will be generated by the system. Although the monitoring system may allow the use of compound events [8] a notation suitable for expressing concurrency constraints is required.

Definition 1 A *concurrency constraint* is an expression determined by the following EBNF specification:

```
expr ::= expr operator expr | '{' expr '}'
    | on (condition) expr
    | policy:action  /* in a role or a relationship*/
    | role:policy:action        /* in a relationship */

operator ::=
        ';'    /* sequential:  a1 ; a2 – a2 must   follow a1 */
    |   '&'    /* and: a1 & a2 – a1 and a2 can be performed in
                  parallel and both must complete */
    |   '|'    /* or: a1 | a2 – perform a1 or a2 */
    |   '#'    /* conflict: a1 # a2 – a1 and a2 cannot overlap */
    |   '||'   /* parallel: a1 || a2 – a1 and a2 may be
                  performed in parallel. Completion of any will
                  continue the execution */
```

Role and policy denote names local to the role or to the relationship.

This notation allows us to express sequences of actions, parallelism and synchronisation in concurrency constraint rules which apply to the policies of a role or of a relationship. For example a physician must first assess the condition of a patient then order a blood test and an antibiotics allergy test. Both must be completed before any antibiotics are prescribed. This can be written in the physician's role as:

```
p1:assess_condition ; {p2:blood_test & p3:allergy_test} ;
p4:prescribe_antibiotics
```

Inside a role a policy is referred to by its unique name local to the role e.g. p1, p2, p3 so as to be able to change a policy in the role without needing to edit the constraints. Concurrency constraints can be translated to complex events [8] which will trigger the execution of activities [13]. This allows us to combine several concurrency constraints in a complex event graph and perform causality analysis on the graph e.g. detection of cycles. A role object maintains a concurrency constraint table where each entry contains a name unique within the table and a constraint expression.

In addition to concurrency constraints Meta-Policies may constrain the group of policies which comprise a role, e.g. the maximum number of patients that one nurse can look after is limited to 10. Furthermore a role refers to the relationships of which it is part and which contain the policies having other related roles as target. We therefore define a role as follows:

Definition 2 A *role (r)* is defined by the MPD and the sets of: policies <p> specifying the obligations and authorisations associated with the position, relationships <rel> in which the role is involved and constraints (concurrency constraints <c> and meta-policies <mp>). r = <mpd,<p>, <rel>, <c>, <mp>> where each element in the sets is designated by a name local to the role (see Figure 2).

For example, the role of a physician in a hospital may contain the obligations towards his patients, authorisations regarding the use and prescription of drugs and his relationships with the nurses, radiologists and administrative staff of the hospital. Note that any policy belonging to a role has the MPD as subject. In the remainder of the paper we will therefore omit the subject of a policy in a role or represent it by the keyword MPD. A role contains references to the relationships it is part of. The set <rel> must therefore correspond to the set of all the relationships in the system which contain the role 'r'.

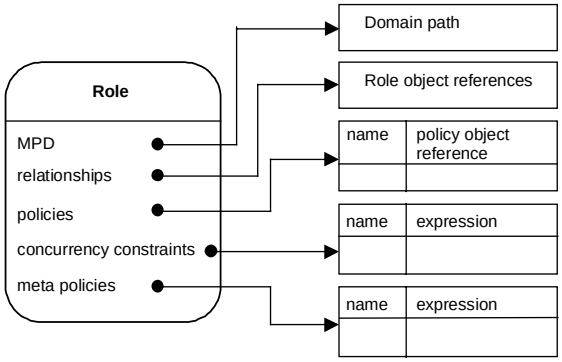

Figure 2 Role Implementation

In the following sections we examine how roles relate to each other and how interaction protocols may be defined.

3.3 Relationships

Relationships between roles define the policies regarding the related roles, e.g. right to assign a task to a role and policies regarding the use of shared resources. However, this is not sufficient since managers interact and co-operate with each other in order to perform their

Example1 Doctor – specialist interaction protocol

Role: doctor	Role: specialist
(1) [request, patient_id] → specialist	(2) request ⇒ [reject, reason] → doctor **or** [results, res] → doctor **or** [req_data, data requested] → doctor
(3) *.req_data ⇒ [data, ...] → specialist	(4) request.*.data ⇒ [results, res] → doctor

tasks. Relationships must therefore specify the protocols for the required interactions between various roles. For example, referring a patient to a specialist must be done according to a pre-established protocol which ensures availability of data and leaves no ambiguity as to whether the patient will be consulted or not by the specialist. Interaction protocols are specified by a set of rules triggered by pattern matching on the type-chain of the incoming messages.

Definition 3 An *interaction protocol* is a set of rules defined according to the following specification:

```
rule ::=
  role regular_expression :: guard ⇒ action_rule

action_rule ::= action | { action } | action 'or' action
                | action ',' action

action ::= object_invocation
         | generate_event(event)
         | message '→' role

message ::= [type_chain, data]
```

where role denotes a name local to the relationship.

Each rule of the protocol is associated with a 'role' of the relationship. Each exchanged message must have a type belonging to a finite Universe of Discourse [14] defined for each interaction protocol. An incoming message triggers one or several of the rules associated with the destination role by matching the regular expressions against the type_chain contained in the incoming message. The type_chain is the sequence of the types of all exchanged messages in the current interaction e.g. request.deny denotes a denial made in response to a request. The guard is a predicate on the contents of the message which must be satisfied before the rule is triggered. The rule indicates a sequence of actions to be performed, reply to be sent or events to be generated. When a human manager is assigned to a role it is necessary to allow the manager some freedom of choice in the course of actions to be taken while still constraining him to remain within the framework of the specified protocol. The human manager can then select between the various matched rules (if several) or choose between the alternatives offered by the use of 'or' in the rules. In the case where automated managers are assigned to a role determinism can be obtained by preventing the use of 'or' in the specification of the rules and defining a simple heuristics for matching the rules e.g. first rule matched. For example the protocol for referring a patient for examination to a specialist may be specified in our notation as shown in Example 1.

Rule (2) allows the specialist to choose between various options: refuse to examine the patient, examine and return results, or request further data. In this last case the protocol prevents the specialist from responding by anything but a result if additional data has been requested. The request for additional data therefore implies that the specialist has accepted the patient for examination (4). The rule matching the incoming request (3) may also be extended to generate an event which can trigger a policy and automatically retrieve some of the information regarding the patient from the database. Each time a new message is sent the type of the message is appended to the type-chain contained in the received message. This enables us to specify regular expressions which take into account the past stages of the interaction. For example request.{req_data.data}3 matches the case where three request_data.data sequences have already occurred. A new rule may in this case allow the doctor to terminate the interaction.

Under the assumption of a finite Universe of Discourse and by using production rules, it is possible to build a commitment calculus which permits assertions to be made about the state of the system and determines the commitments of the participants in the interaction from the messages exchanged. A large amount of work exists in this area based on the Speech Acts theory [15]. It is not our intention to produce yet another normative speech system, but it is worth pointing out that such a facility can be built for the purposes of office automation using the given interaction rules.

Both the interaction protocol rules and the policies pertain to a role i.e. they are associated with a role within the relationship. Relationships may be constrained in the same way as roles by concurrency constraints and meta-policies for which they define a scope of validity.

Definition 4 A *relationship (rel)* is defined by the set of roles <r>, policies <p>, interaction protocols <ip> and constraints (<c> and <mp>) defining the behaviour of the related parties. rel = <<r>, <p>, <ip>, <c>, <mp>> where

each element in the set is designated by a name local to the relationship.

Several types of relationships can be defined in this way. Supervision of work can be specified by obligations to provide reports, checks on completed tasks and protocols for requesting authorisations. Contractual relationships are defined by the obligations of each of the contractual parties, authorisations regarding the use of shared resources and interaction protocols for negotiation and exchange of documents. A wide range of relationships can therefore be modelled giving a more accurate and flexible representation of the organisational structure. In particular, organisations with decentralised management can be modelled in addition to the centralised hierarchical structure. When a role of the relationship (<r> set) is a target of one of the policies (e.g. assignment of a task), the role is designated by its unique name local to the relationship e.g. doctor, specialist. This indirection is necessary in order to be able to replace a role in the relationship without changing the policies or other components of the relationship. Figure 3 summarises the components of our Role framework.

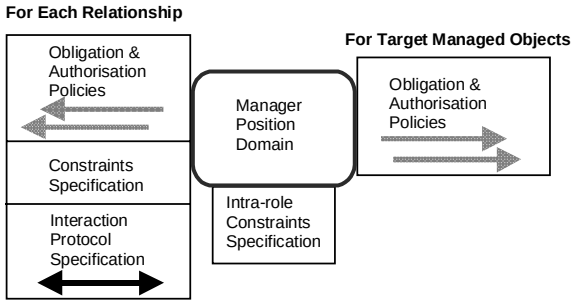

Figure 3 Related Role

4. Classes and Templates

An organisation may contain large numbers of roles with few differences between them. Furthermore, each role may be part of a large number of relationships. We introduce classes and templates in order to reduce the number and complexity of the specifications. For example a nurse role class can be specified and used to create the nurse-instance roles for wards 3,4 and 10. Each instance may then be customised for any particular task relating to a specific ward and a specific person assigned to each role. In this section we define the role object model and examine its uses. The definition of role classes is based upon policy templates (which are specifications of duties and rights independent of subject, target or both).

4.1 Policy Templates

Policy templates are used in order to provide the reuse of the policy specifications according to their domain of application.

Definition 5 A *policy template* uses variables to represent subjects and/or targets i.e. specifies the policy actions and constraints which can be reused for different subjects and targets.

The instantiation of a policy object from a policy template is done by specifying the subjects and targets. For example in a hospital a policy template such as the one below may be specified (S and T represent variables). Note the use of the constraint to limit the applicability of the policy to particular object instances within the target domain T.

/ subjects are authorised to administer analgesics when the temperature of the target is between 37 and 38.5 */*

A+ S { administer(analgesics) } x:T
when (x.temp > 37) && (x.temp < 38.5)

The following policy authorising a nurse to administer analgesics to lung-disease patients may then be created from the above template by assigning values to S and T.

A+ @/personnel/nurses { administer(analgesics) }
x:@/patients/lung-diseases
when (x.temperature > 37) && (x.temperature < 38.5)

A policy template may not inherit from another policy template since the components of a policy (actions, condition, trigger, etc.) are closely related and cannot be combined by inheritance. The policy will maintain a reference to the template from which it has been created in addition to the references maintained to the policies it has been refined from. When a policy template is instantiated from a role class, only the target has to be specified since the subject is determined by the MPD. Similarly, upon instantiation of a relationship the subject of a policy is the MPD of the role it is associated with.

4.2 Defining Role Classes

A role class groups specifications of the policy templates for the duties and rights of a generic role in the organisation e.g. nurse, engineer, marketing manager. When a role instance is created, a MPD for the role is then created so a manager cannot be assigned to a role class, only to a role instance. The role class contains only policy templates, not instances. Consider the example of the nurse role class, containing a set of policy templates which may have some undefined targets. The nurse class may contain the following templates:

/ the nurse is authorised to access the drugs database */*

pt_1 **A+** D { read(), search(), update() }
@/software/databases/drugs_db

/ nurses must monitor their patients */*

pt_2 **O+** D { monitor() } P

When the role is instantiated the MPD can be assigned to the variable D of template pt_1 which is then fully specified as the same target domain is used for all nurse instances. However pt_2 also needs a target domain representing the specific patients for which the nurse instance is responsible, to be assigned to variable P. Note that instead of specifying the pt_1 policy template, a global policy instance (which is not part of the role) with a subject referencing a domain containing all instances of the nurse role could have been specified. The disadvantage of this approach is that the system must be constrained to include every nurse role in a given domain which is very difficult to implement and to check. If multiple instances of a particular policy template are required in a role, then the role class must specify a different local name for each reference to the policy template. This situation is however unlikely to occur because, within the role, policies created from the same template differ only in their targets which can be composed into a single domain scope expression to define all the targets e.g. domain_A + domain_B - domain_C. The concurrency constraints defined in a role class will apply, without change, to all the instances of the class. This is because the concurrency constraint refers to policies by their local name in the role and only the targets of the policy change between the policy instance and the policy template.

Concurrency constraints may also be specified for policy templates and refer to policies by their local name within the roles or relationships. They reference only the activities within a policy, so they can apply to either policy objects or templates. Consider the following policy templates.

/ the nurse must administer analgesics when the temperature of the target is > 37 */*

pt_3 **O+ on** x.temperature > 37
D { administer(analgesics) } x:T

/ the nurse must update the database when drugs have been administered */*

pt_4 **O+** on administer_drugs D { update }
@/software/databases/drugs_db

The following constraint ensures that the drugs database is updated after drugs have been administered.

c_1 pt_3:administer(analgesics) ; pt_4:update

Finally a role class may reference relationship classes defined as described in the following section. The reference to a relationship class specifies that an instance of the role class cannot be created without the corresponding relationship being instantiated. For example a nurse role may not be created without creating a relationship with a head of ward role. Note that only required relationships must be specified. Additional relationships, for example with the physicians, may be added at a later stage.

Definition 6 A *role class (r*)* is defined by an MPD variable, a set of policy templates, a set of intra-role concurrency constraints, a set of meta policies and a set of relationship classes. r* = <MPD, <p*>, <c>, <mp>, <rel*>> where each element in the sets is designated by a local name.

An instance of the role class can be created by creating the MPD, instantiating policy objects by specifying all undefined targets for each of the elements in the <p*> set, and assigning references to the relationships in the <rel*> set. Concurrency constraints apply to policies or policy templates. The meta-policies can be used to prevent policies which do not satisfy the meta-policy constraints from being created.

4.3 Implementing inheritance

Single inheritance implements a specialisation of role classes. For example a role class can be specified for a specialised nurse or a paediatric nurse which inherit from the nurse role class (Figure 4). New duties and rights can then be specified for the specialised nurse that do not apply for the nurse. Furthermore, a sub-class must be able to override or cancel policy templates of a super-class e.g. a specialised nurse may not have a duty to monitor all patients but deal only with specific cases (Figure 4 - policy pt_2). Inheritance can be implemented by maintaining a reference to the super-class object and new policy templates must have different names from those in the super-class. If a new policy template or constraint has the same name as one from the super-class, the inherited one will be overridden. Concurrency constraints are evaluated in the namespace of the sub-class so policy names will be replaced with the local reference overriding inherited ones. However, overriding policy templates may introduce inconsistencies if the new policy does not contain the actions referred to by the concurrency constraints. These inconsistencies may be detected by an automated process of the role editor but changes to the concurrency constraints will have to be done manually.

Composing role classes by **multiple inheritance** may also be a desirable feature. For example a paediatric nurse

can inherit rights and duties from a nurse role class and a generic_childcare role class (see Figure 4). However, multiple inheritance introduces the problems arising from multiple super-classes having policies with the same name. For example in Figure 4, both the nurse and the generic_childcare role classes define a policy template with the name pt_2. A standard solution is to unify the name spaces in the subclass and assign a precedence order to the super-classes e.g. the textual order in the class inheritance list. It is then possible to override the unified policy names with new ones defined in the sub-class, as for single inheritance. We intend to implement multiple inheritance by combining the specifications of the super-classes rather then having policies with the same name override each other. In this case the name spaces of the super-classes are kept disjoint. Referring to an inherited policy template, whether to override it or within a new concurrency constraint must then be done by prefixing the name of that template with the name of the super-class from which it was inherited e.g. nurse.pt_2 (Figure 4). Inherited concurrency constraints have to be evaluated within the context of the class from which they have been inherited plus all overridden policy templates.

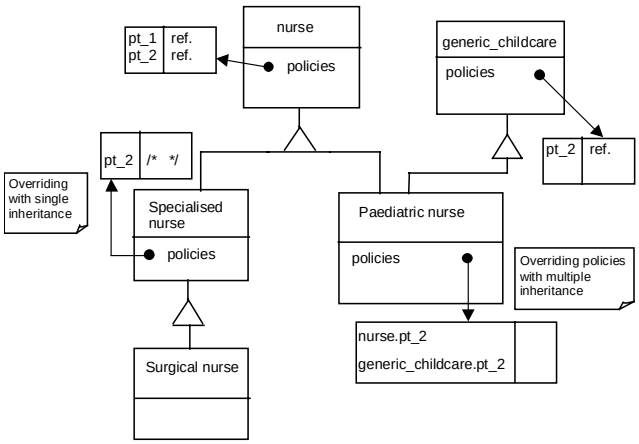

Figure 4 Role class inheritance graph

Super-classes also maintain references to the sub-classes which inherit from them and classes maintain references to the objects which have been instantiated from them. This permits us to define skills [16]. Skills may be associated with roles outlining their basic capabilities. A trader service may then be used to locate roles which have specific skills. It is thus possible for the trader to browse the inheritance graph and determine the roles which have for example, the skills of a nurse. The definition and implementation of skills requires however further study.

A class may not be changed once sub-classes or instances have been created from it. We may in the future relax this constraint and propagate changes to subclasses and to instances. This is possible because references to subclasses are maintained as mentioned above.

4.4 Relationship Classes

A relationship class has a set of elements each of a given role class and a possibly empty set of roles e.g. the relationship class with a unique head of personnel role. A relationship class may therefore contain the policy objects pertaining to the roles and a set of policy templates pertaining to the role classes.

Definition 7 A *relationship class (rel*)* is defined by a (possibly empty) set of roles <r>, a set of role classes <r*>, a set of interaction protocol rules <ip>, and a set of constraints (concurrency <c> and meta-policies <mp>). The relationship class also contains a set of policies <p> (each one of them being associated with a role) and a set of policy templates <p*> (each one of them being associated with a role class). Interaction protocol rules may be associated with either a role or a role class. rel* = <<r>, <r*>, <p>, <p*>, <c>, <mp>, <ip>>

Every instance of a relationship class containing role instances will be a new relationship with those role instances. Within a relationship class the concurrency constraints may refer to items in either the <p> or the <p*> set. A relationship class can be instantiated by assigning each of the references in the sets <p*> and <r*>. The named elements of the <r*> set act as place holders i.e. upon instantiation of the relationship class a role instance must be assigned to that name. Further the role instance must be an instance of the class which is associated with the name in the relationship class.

Consider for example a relationship class between a nurse role class, a physician role class and a head of ward role (Example 2). The relationship contains the policy templates obligating the nurse to report every day to head of ward (np1). The physician has the obligation of specifying treatments for the patients (pp1) and the head of ward is responsible for assigning duties to the nurse, scheduling the activities and ordering the necessary drugs (hp1, hp2, hp3). Furthermore, an interaction protocol is provided which enables the doctor to request that a special treatment be performed (ip1). The request has to be sent to the head of ward which can decide to reject it or to assign a nurse (ip2). The reply must then be sent to both the nurse and the physician. The nurse must notify both the physician and the head of ward (ip3) when the treatment has been completed. c_1 constrains the head of ward to assign duties only after the care treatment has been specified by the physician, a schedule has been elaborated and drugs have been ordered.

Example 2 Relationship class

R1: nurse (role class)	*R2: physician (role class)*	*R3: head of ward (role)*
Policies		
np1: **O+** every 1*day { report} R3	pp1: **O+**{specify_care } patients	hp1: **O+** {assign_duties} R1 hp2: **O+** {generate_schedule_of_activities} hp3: **O+** {order_drugs } pharmacy
Interaction protocols e.g. special treatment		
ip3: *.appoint => [confirm, ...] → R2, R3	ip1: [request, ...] → R3	ip2: request => [reject,...] → R2 **or** [assign, ...] → R1, R2
Concurrency constraints		
c_1: R2:p1:specify_care ; {R3:p2:generate_schedule & R3:p3:order_drugs } ; R3:p1:assign_duties		

Single inheritance and overloading occur for the relationship classes in much the same way as for the role classes. For example the relationship between a surgical nurse and a surgeon may inherit from the relationship between a nurse and a doctor and add special interaction protocols for urgent interventions (Figure 5).

Overloading of a role or a role class in a relationship is restricted to one of the sub-classes of the roles in the relationship super-class. For example relationship2 could have been between a surgeon and a specialised nurse but not between a surgeon and a nurse or any other role class which is not a sub-class of a specialised nurse. In this respect the role class hierarchy implements a type/sub-type mechanism for the relationship classes. Multiple inheritance is not allowed because a direct combination of two relationships by inheritance is undefined.

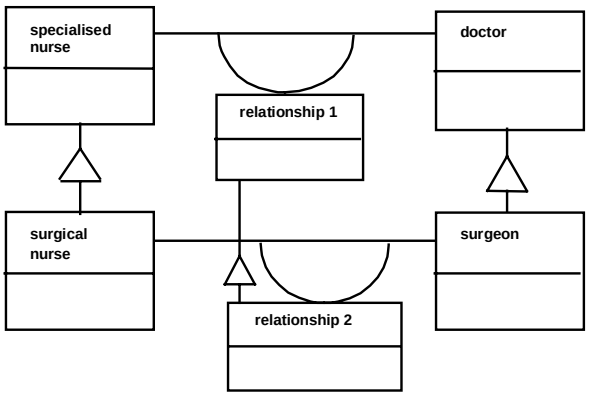

Figure 5 Role and relationship inheritance

5. Consistency

Consistency of the role framework is ensured by various checks on the role and relationship specifications. The role editor will therefore use a set of tools for analysing and detecting conflicts or inconsistencies in the specification. We have already implemented a tool for off-line detection of modality conflicts which uses the principle of domain nesting [17] for giving precedence to some policies and automatically resolving some conflicts.

Modality conflicts are inconsistencies in the policy specification which may arise when two or more policies with modalities of opposite sign refer to the same subjects, actions and targets. This occurs when there is a triple overlap between the sets of subjects, targets and actions, and so can be determined by syntactic analysis of polices. There are three types of modality conflicts:

- **O+/O-** the subjects are both required and required not to perform the same actions on the target objects.
- **A+/A-** the subjects are both authorised and forbidden to perform the actions on the target objects.
- **O+/A-** the subjects are required but forbidden to perform the actions on the target objects (obligation does not imply authorisation in our case).

Obligations in our model do not imply authorisation and we assume a negative default policy i.e. everything is forbidden. In this way any action present in an obligation policy and not specifically authorised by an authorisation policy will be detected as a conflict.

A second type of conflict refers to the consistency between what is contained in the policies i.e. which subjects, targets and actions are involved and external criteria such as limited resources or the overall policies of

the organisation. An example of this type of conflict arises from the principle of separation of duties [18] e.g. the same manager cannot authorise payments and sign the payment cheques. These conflicts are **application-specific** and must be specified using meta-policies. Several types of application-specific conflicts such as conflict of priorities for resources, conflict of duties, conflict of interests, multiple managers conflict and self-management conflict have been identified in [12] and classified according to the overlaps between the subject, action and target sets. Further details on the conflict analysis for management policies are given in [3]. We have been experimenting with meta-policies written in Prolog but will have to define a notation more specific to the policies.

Concurrency constraints may also exhibit inconsistencies; for example defining a cyclic interdependency between the activities. These may be detected by analysing the event graph which can be generated from the constraints. Any presence of cycles in the graph will reveal an inconsistency.

Interaction protocols may suffer from incompleteness rather than inconsistency. Their specification by rules may lead to confusion regarding 'who sends which message' or 'is there a rule triggered by the sent message'. Because we assume a finite Universe of Discourse it is possible to initially specify a protocol by a state transition network [19]. The initial set of rules can then be deduced by considering the regular expressions which match the various states [13]. By providing a graphical interface to the state transition network specification is made easier and completeness problems can be avoided.

6. Related Work

Sandhu et al. introduce four different models for Role Based Access Control (*RBAC*0-3) [20]. In *RBAC*0 a role is a mapping between a set of users and a set of permissions. A set of users and a set of permissions are thus associated under a named role e.g. all the nurses in the hospital have the role 'nurse' and the access rights associated with it. A user is allowed to combine several roles in a single session. *RBAC*1 introduces role hierarchies as a means of inheriting access rights from one role to another. Senior roles inherit the access rights of more junior roles e.g. a physician inherits the access rights of a general health-care provider. This approach introduces undesired complexities when some of the access rights must be kept private to a more junior role e.g. a radiologist cannot issue prescriptions and thus should not inherit this access right from a physician. This leads to the proliferation of 'virtual' roles which do not correspond to positions in the organisation but are merely used to define common permissions which can be inherited. *RBAC*2 introduces constraints on the role structure which bear some similarity to our meta-policies. Finally *RBAC*3 combines the features of the previous three models. This work deals only with access rights and not duties which somewhat simplifies the problem. Permissions always associate actions with target objects so they cannot define policy templates. This causes difficulties instantiating multiple role instances referring to different target objects.

Singh and Rein describe a role structure based on activities and interactions [21],[22],[23],[24]. They define the role in terms of available methods and relationships to 'partner' objects. Moreover, they identify a set of conflicting objects, temporal ordering constraints and constraints on the activities for each role. The interactions between roles are based on Petri-Net modelling although a higher-level graphical notation is defined. The model has a sound operational semantics based on PetriNets but does not take into account access control issues and does not provide any means for inheritance or reuse of the specification.

Skarmeas defines roles for multi-agent systems [16]. An elementary role is defined as a collection of complex tasks and a hierarchy of roles is introduced which reflects the organisational task decomposition. Roles relate to each other by contracts which may contain sub-roles and references to other contracts. This work uses roles as a structuring tool for tasks which are assumed to be hard-coded and available to the agent. It ignores problems related to constraints, inheritance or concurrency.

7. Conclusions

This paper has introduced policies, roles and relationships for specifying enterprise viewpoint concepts which can be implemented by computational objects thus permitting automation of management. Roles identify the duties and authorisations of the individuals which are assigned to them. It is possible to assign and withdraw managers from roles without changing the specification of the role thus enabling rapid and flexible organisational change. Auditing of the role specification can be achieved by analysing the policies, with their explicit subject and target domains, to determine the duties and permissions of an individual assigned to multiple roles or to determine who has responsibility for or access rights to an object.

We have defined the specification of relationships between roles which include the protocols for required interactions. It is thus possible to model various types of relationships such as peer-to-peer, supervision and contractual arrangements. Roles and relationships can be

used to describe and analyse the organisational structure and to implement policies and procedures for enforcing security and ensuring quality.

There are many roles and relationships in an organisation. Similar specifications can be grouped in classes permitting re-use of the specifications and incremental design. We have shown what information is needed to create role and policy instances from their classes. Our work has shown that relying only on instance inheritance is not practical. It is essential to be able to parameterise instances. We are currently implementing the role object model and the role and relationship editor using a CORBA-based distributed programming environment. Although the current role editor offers the basic support needed for defining roles, relationships, policies and their associated classes it does not yet cater for the automatic detection of name clashes or checks for consistency of specifications. Also it does not yet offer transparent visualisation of inherited elements.

A conflict detection tool for modality conflicts has been implemented and some work on the specification and validation of meta-policies has been done [3]. Further work is needed along this line in particular for defining a meta-policy notation (we have used Prolog up to now). Although initial grammars of both concurrency constraints and interaction protocol rules have been specified, further work is in progress for the implementation of the role agents which interpret obligation policies.

We intend to investigate the applicability of software engineering goal refinement tools and techniques for policy refinement and its bearing on the role object model. In particular the task decomposition and the assignment of sub-tasks to roles should be tracked to and from the roles and relationships for auditing purposes. In addition further work is needed on checking consistency of policy specifications and to determine what actions are performed when an event is triggered. This will form part of a role and policy specification toolset.

8. Acknowledgements

We gratefully acknowledge financial support for the EPSRC RoleMan project (GR/K 37512), from Fujitsu Network Systems Laboratories for the Pro-Active Role Based Management for Distributed Services project and from British Telecom for the Management of Multimedia Networks project. We are grateful to Stephen Crane and Nat Pryce for many comments which have improved this paper. We also acknowledge the contribution of our colleagues to the concepts described in this paper.

9. References

[1] B. J. Biddle, *Role Theory, Expectations Identities and Behaviour*, Academic Press Inc, 1979.

[2] B. J. Biddle and E. J. Thomas, Eds, *Role Theory: Concepts and Research.* New York, Robert E. Krieger Publishing Company, 1979.

[3] E. C. Lupu and M. S. Sloman, "Conflict Analysis for Management Policies", IFIP/IEEE International Symposium on Integrated Network Management (IM formerly known as ISINM 97), San Diego, Chapman & Hall, 1997.

[4] R. Reddy, et al., "ARTEMIS: A Research Testbed for Collaborative Health Care Informatics", IEEE WET-ICE, Morgantown, pp. 60-65, IEEE Computer Society Press, 1993.

[5] M. Fowler, et al., "Using Object-Oriented Analysis, Design and Implementation Techniques in the Clinical Domain", *Institute of Physics Publishing Bulletin* 1993. *Observations and Measurement. Report on Object Analysis and Design*, 2,3, (1995), pp. 20-24,37.

[6] M. S. Sloman, "Policy Driven Management for Distributed Systems". *Journal of Network and Systems Management*, **2**(4): 333-360, Plenum Press Publishing, 1994.

[7] M. S. Sloman and K. P. Twidle, "Domains: A Framework for Structuring Management Policy", In *Network and Distributed Systems Management*. Sloman M. ed., Addison Wesley, 433–453, 1994.

[8] M. Mansouri-Samani and M. S. Sloman, "GEM - A Generalised Event Monitoring Language for Distributed Systems", *IEE/IOP/BCS Distributed Systems Engineering Journal*, **4(2)**, June 1997.

[9] D. Marriott and M. Sloman, "Management Policy Service for Distributed Systems", *IEEE* Third Int. Workshop on Services in Distributed and Networked Environments (SDNE'96), pp. 2-9, Macau , June 1996.

[10] D. Marriott and M. Sloman, "Implementation of a Management Agent for Interpreting Obligation Policy", IEEE/IFIP Distributed Systems Operations and Management (DSOM 96), L'Aquila (Italy), 1996.

[11] N. Yialelis and M. Sloman, "A Security Framework Supporting Domain-Based Access Control in Distributed Systems", IEEE ISOC Symposium on Network and Distributed Systems Security'96, San Diego, pp. 26-34, Feb. 1996.

[12] J. D. Moffett and M. S. Sloman, "Policy Conflict Analysis in Distributed System Management", *Ablex Publishing Journal of Organisational Computing*, **4**(1): 1-22, 1994.

[13] E. C. Lupu and M. S. Sloman, "Towards a Role Based Framework for Distributed Systems Management", *Journal of Network and Systems Management*, **5**(1), Plenum Press Publishing, 1997.

[14] P. de Greef, et al., "Towards a Specification Language for Cooperation Methods", 16th German AI-Conference, GWAI'92, Berlin, Springer-Verlag, 1992.

[15] J. R. Searle, *Expression and Meaning-Studies in the Theory of Speech Acts*, Cambridge University Press, 1979.

[16] N. Skarmeas, "Organisations through Roles and Agents", International Workshop on the Design of Cooperative Systems (COOP'95), Antibes-France, 1995.

[17] A. Heydon, "Miro: Visual Specification of Security", *IEEE Transactions on Software Engineering*, **16**(10): 1185-1197, IEEE Press, 1990.

[18] D. C. Clark and D. R. Wilson, "A comparison of Commercial and Military Computer Security Policies", IEEE Symposium on Security and Privacy, 1987.

[19] T. Winograd, "A Language/Action perspective on the Design of Cooperative Work", In *Computer Supported Cooperative Work: A book of readings.* I. Greif ed., Morgan Kaufmann Publishers, 1988.

[20] R. S. Sandhu, et al. "Role-Based Access Control Models", *IEEE Computer* **29**(2): 38-47, IEEE Press, 1996.

[21] B. Singh, "Interconnected Roles (IR): A Coordination Model", Technical Report CT-084-92, MCC, 1992.

[22] B. Singh and G. L. Rein, "Role Interaction nets (RINs): A Process Description Formalism", Technical Report, CT-083-92, MCC, 1992.

[23] G. L. Rein, "Collaboration Technology for Organisation Design", Twenty-Sixth Annual Hawaii International Conference on System Sciences, 1993.

[24] G. L. Rein, et al., "The Grand Challenge: Building Evolutionary Technologies", Twenty-Sixth Annual Hawaii International Conference on System Sciences, Maui (Hawaii), 1993.