Conflicts in Policy-Based Distributed Systems Management

Emil C. Lupu, Member, IEEE, and Morris Sloman, Member, IEEE Computer Society

Abstract—Modern distributed systems contain a large number of objects and must be capable of evolving, without shutting down the entire system, to cater for changing requirements. There is a need for distributed, automated management agents whose behavior also has to dynamically change to reflect the evolution of the system being managed. Policies are a means of specifying and influencing management behavior within a distributed system, without codifying the behavior into the manager agents. Our approach is aimed at specifying implementable policies, although policies may be initially specified at the organizational level (e.g., goals) and then refined to implementable actions. We are concerned with two types of policies. Authorization policies specify what activities a manager is permitted or forbidden to do to a set of target objects and are similar to security access-control policies. Obligation policies specify what activities a manager must or must not do to a set of target objects and essentially define the duties of a manager. Conflicts can arise in the set of policies. For example, an obligation policy may define an activity which is forbidden by a negative authorization policy; there may be two authorization policies which permit and forbid an activity or two policies permitting the same manager to sign checks and approve payments may conflict with an external principle of separation of duties. Conflicts may also arise during the refinement process between the high-level goals and the implementable policies. The system may have to cater for conflicts such as exceptions to normal authorization policies. This paper reviews policy conflicts, focusing on the problems of conflict detection and resolution. We discuss the various precedence relationships that can be established between policies in order to allow inconsistent policies to coexist within the system and present a conflict analysis tool which forms part of a role-based management framework. Software development and medical environments are used as example scenarios in the paper.

Index Terms—Obligation policy, authorization policy, meta-policy, policy conflict, conflict resolution, management roles.

1 INTRODUCTION

Distributed systems may contain a large number of objects and potentially cross organizational boundaries. New components and services are added or removed from the system dynamically, thus changing the requirements of the management system over a potentially long lifetime. There has been considerable interest recently in policy-based management for distributed systems [51], [5], [26], [19]. A Policy is information which can be used to modify the behavior of a system. Separating policies from the managers which implement them permits the modification of the policies to change the behavior and strategy of the management system without recoding the managers. The management system can then adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system. We are concerned with two types of policies. Authorization policies are essentially security policies related to access-control and specify what activities a subject is permitted or forbidden to do to a set of target objects. Obligation policies specify what activities a subject must or must not do to a set of target objects and define the duties of the policy subject. We permit the specification of both positive and negative authorization policies and require explicit authorization, i.e., nonauthorized invocations are forbidden. An overview of our policy notation is given in Section 2.2.

The subject of a policy specifies the human or automated managers to which the policies apply and which interpret obligation policies. The target of a policy specifies the objects on which actions are to be performed. Domains are a means of grouping objects and are similar to file system directories. They are described in more detail in Section 2.1 below. The subject or target of a policy is expressed as a domain of objects and the policy applies to all objects in the domain; so a single policy can be specified for a group of objects [52]. This helps to cater for large-scale systems in that it is not necessary to define separate policies for individual objects in the system, but rather for groups of objects.

In a large distributed system there will be multiple human administrators specifying policies which are stored on distributed policy servers. Policy conflicts can arise due to omissions, errors or conflicting requirements of the administrators specifying the policies. For example an obligation policy may define an activity a manager must perform but there is no authorization policy to permit the manager to perform the activity. Conflicts can also arise between positive and negative policies applying to the same objects (which we refer to as modality conflicts). In general, whenever multiple policies apply to an object there is a potential for some form of conflict, but it is essential that multiple policies should apply in order to cover the diversity of management functions and of management domains. There may be different policies relating to security, monitoring, or configuration which apply to a set
of objects reflecting different management functions which
may be performed on the objects. Similarly, the policies
specified for the network, subnetwork, and workstation
domains will all propagate to the network objects inside
the workstation.

Many policies specified for the management of a large
system specify exceptions to more general policies. System
administrators are typically permitted to reboot computer
systems while ordinary users are prohibited from perform-
ing such actions. It is not always desirable to eliminate the
conflict by rewriting the policies or changing the mem-
bership of the domains to which policies apply. As automated
managers cannot enforce conflicting policies, a precedence
relationship must be established between the policies in
order to resolve the conflicts.

In this paper, we review the conflicts which may arise
between management policies and describe the tools we
have developed for analyzing policy specifications to
determine conflicts. We use roles as the means of grouping
policies related to a particular manager position and then
managers can be assigned or removed from the position
without changing the policies [23]. We also define the
relationships between roles with regard to the use of shared
resources or with regard to the organizational structure,
e.g., a departmental manager role will have the right to
assign tasks to the section manager role. A large-scale
distributed system will have very large numbers of objects
and policies distributed around the system, so the conflict
detection cannot be centralized but also has to be
distributed. Our use of roles and inter-role relationships
provides a scope for the conflict detection and helps to limit
the number of policies that have to be examined in order to
determine conflicts. This paper focuses on techniques and
tool support for off-line conflict detection and resolution,
although some conflicts can be detected only at run-time.

Policies are interpreted by automated manager agents
and so the behavior of the agents can be modified
dynamically by changing policy rather than recoding. We
use the term "agent" to refer to an automated component
which interprets policies. The policies thus provide a
constrained form of "programming" of automated agents
to change management strategy without shutting down the
management system. As management activities can have a
drastic impact on the system being managed, it is important
to determine and resolve policy conflicts so that the
automated management is able to perform correctly. The
policies can also apply to humans, for example the roles
related to a collaborative software development team. Our
policy notation and role framework could be used to specify
the rights (authorizations) and duties (obligations) of
members of the team. It is useful to be able to determine
policy conflicts within a single role or between roles by
analysis of the policies rather than relying on human
initiative to resolve the conflicts when they occur.

The work presented in this paper stems from research on
software paradigms for the management of distributed
systems. However, most of the principles outlined here also
apply to the engineering of large software applications.
Authorizations policies are often embedded in database
management systems in order to ensure the privacy of
information [22]. The opportunity of downloading and
running programs (Java, SafeTcl, etc.) from sources with
varying degrees of trust, requires the host application to
configure access control according to security policies
which may be explicit or implicit. Minsky has extensively
studied the use of permission and prohibition rules for
specifying laws with which application components must
comply [34]. Obligation policies can be either used in
conjunction with authorizations in order to ensure the
integrity of the system [33] or to declaratively specify the
actions a component must initiate in response to changes in
its internal state or environment.

In Section 2 of this paper, we give more details of the
domains, policies, and roles which form our management
framework. Section 3 discusses the type of inconsistencies
and the policy conflicts we need to detect. In Section 4 and
Section 5, we explain our approach to conflict detection, and
conflict resolution based on policy precedence relations-
ships. Section 6 describes the prototype tools the authors
have implemented. Relationship to other works are covered
in Section 7, followed by conclusions and further work in
Section 8. In this paper, the authors do not address
inconsistencies that may arise as a result of partial failures
within a distributed system.

2 MANAGEMENT FRAMEWORK

The main components of our management framework are domains for grouping objects, a policy service to support
the specification and storage of policies and roles to reflect
the organizational structure, responsibility, and relationships
between management positions.

2.1 Domains

Domains provide a flexible means of partitioning the objects
in a large system according to geographical boundaries,
object type, management functionality, responsibility, and
authority or for the convenience of human managers. In
many cases, domains are used to group objects in order to
apply a common policy to a set of objects, e.g., in a
department within a company. Membership of a domain is
explicit and not defined in terms of a predicate on object
attributes. A domain does not encapsulate the objects it
contains but merely holds references to object interfaces. A
domain is thus very similar in concept to a file system
directory but may hold references to any type of object,
including a person. A domain, which is a member of
another domain, is called a subdomain of the parent
domain. A subdomain is not a subset of the parent domain,
in that an object included in a subdomain is not a direct
member of the parent domain, but is an indirect member
(c.f., a file in a subdirectory is not a direct member of a
parent directory). An object or subdomain may be member
of multiple parent domains. For example, in Fig. 1, the two
"bean people" and subdomain E are members of both B and
C domains which, therefore, overlap. We permit cyclic
structures within the domain hierarchy, as it is easier to deal
with them in domain traversal algorithms than to try to
prevent them. Details of domains are described in [51], [52].

Path names are used to identify domains, e.g., domain E
can be referred to as /A/B/E or /A/C/E as an object may have
different local names with multiple parent domains, where "/" is used as a delimiter for domain path names. Policies normally propagate to members of subdomains, so a policy applying to domain B will also apply to members of domains D and E. Domain scope expressions can be used to combine domains to form a set of objects, for applying a policy, using union, intersection, and difference operators, e.g., a scope expression \( @/A/B + @/A/C - @/A/B/E \) would apply to members of B plus C but not E, and \( @/A/B \land @/A/C \) applies only to the direct and indirect members of the overlap between B and C. The "\(@\)" symbol selects all nondomain objects in nested domains.

An advantage of specifying policy scope in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies.

2.2 Policies

In this section, we give an overview of the notation used to specify policies [28], [22]. The notation is essentially aimed at specifying policies which are interpreted by automated agents, but can also be used to specify high-level abstract policies or goals that could only be interpreted by humans. As stated in Section 1, the policies are interpreted rather than compiled into the code of agents, so can be changed dynamically. The notation is precise and can be analyzed for conflicts using tools, but it is not based on a well-known logic. Implementable policies are directly interpreted by automated manager and access control agents, which are (potentially) distributed, so we do not use logical deduction in order to analyze the state of the system. Our notation should not be confused with Deontic Logic as our authorizations are independent from obligations. The interdefinability axiom in Standard Deontic Logic, where permissions are defined in terms of obligations, i.e., \( P_x = \text{Def.} \neg O \rightarrow \neg x \), does not apply to our notation.

Authorization policies define what activities a subject can perform on a set of target objects and are essentially access control policies to protect resources from unauthorized access. Constraints can be specified to limit the applicability of both authorization and obligation policies based on time or values of the attributes of the objects to which the policy refers.

\[
\begin{align*}
\text{x1 A+ @/project-managers } & \{ \text{defer(); activate() } \} \\
& \text{x@/tasks/modification_requests} \\
& \text{when } (x.\text{status} = \text{approved})
\end{align*}
\]

Project managers are authorized to defer or activate modification requests that have been approved. The ";" is used to separate the permitted actions. Note the use of the constraint to limit the scope of applicability of the policy to objects in the target domain with status = approved.

\[
\begin{align*}
\text{x2 A- @/test-engineers } & \{ \text{commit(); edit() } \} /\text{repository/db} \\
& \text{when } (20:00 < \text{time}) \text{ or } (\text{time} < 07:00)
\end{align*}
\]

Test engineers are forbidden to commit new changes or edit the repository database between the hours of 8:00 p.m. and 7:00 a.m. the following day, i.e., a time-based constraint. The ";" is used to separate the forbidden actions. Note, that if there is a default negative authorization policy, whereby all actions are forbidden unless explicitly authorized, the negative authorization in x2 could be converted into a positive authorization with a constraint when 07:00 < time < 20:00.

Obligation policies define what activities a manager or agent must or must not perform on a set of target objects. Positive obligation policies are triggered by events.

\[
\begin{align*}
\text{x3 O+ on new_request(mrl) @/project1/analysts} \\
& \{ \text{investigate(mrl); propose_solution(mrl) } \} \\
& /\text{project2/tasks/modification_requests;}
\end{align*}
\]

This positive obligation policy is triggered by an external event signaling that a new modification request has been issued and obliges the analysts to investigate and then propose a solution to the modification request. The ";" is used to separate a sequence of actions in an obligation policy.

\[
\begin{align*}
\text{x4 O+ at 01:00 /archiver } & \{ \text{backup() } \} /\text{repository/db}
\end{align*}
\]
This positive obligation policy is triggered by an internal event—every night at 1:00 a.m.—for the archiver to backup the repository database.

\[ x5 \text{-} n: @/test-engineers \{ DiscloseTestResults() \} @/analysts + @/developers \]

when n.testing_sequence == in-progress

This negative obligation policy specifies that test engineers must not disclose test results to analysts or developers when the testing sequence is being performed by that subject is still in progress, i.e., a constraint based on the state of subjects.

The general format of a policy is given below with optional attributes within brackets (the braces and semicolon are the main syntactic separators). Some attributes of a policy such as trigger, subject, action, target or constraint may be comments (e.g., / this is a comment */) in which case the policy is considered high-level and not able to be directly interpreted.

identifier mode [trigger] subject "["action"]" target [constraint] [exception] [parent] [child] [xref] ";"

The identifier is a label used to refer to the policy. The mode of the policy distinguishes between positive obligations (O-), negative obligations (O+), positive authorizations (A+), and negative authorizations (A-).

The trigger only applies to positive obligation policies. It can specify an internal timer event using an at clause, as in x4 above, or an external event using an on clause, as in x3 above, where the now.request event passes a parameter (mil) to the agent. Examples of external events are a temperature exceeding a threshold or a component failing. These are detected by a monitoring service. The policy notation only specifies simple events as a generalized monitoring service can be used to combine event sequences to generate simple events [27].

The subject of a policy, defined in terms of a domain scope expression, specifies the human or automated managers and agents to which the policies apply and which interpret obligation policies. The target of a policy, also defined in terms of a domain scope expression, specifies the objects on which actions are to be performed. Security agents at a target’s node interpret authorization policies and manage agents in the subject domain interpret obligation policies.

The actions specify what must be performed for obligations and what is permitted for authorizations. It consists of method invocations or a comment and may list different methods for different object types. Multiple actions in an authorization policy indicate the set of actions or operations which are permitted or forbidden. Multiple actions in a positive obligation policy imply that they are performed sequentially after the policy is triggered.

The constraint, defined by the when clause, limits the applicability of a policy, e.g., to a particular time period as in policy x2 above, or making it valid after a particular date (when time > 1/June/1999). In addition, the constraint could be based on attribute values of the subject (as in policy x5 above) or target objects. In x5, the label n, prepended to the subject, is referenced in the constraint to indicate a subject attribute. Constraints must be evaluated every time an obligation policy is triggered or an authorization policy is checked to see whether the policy still applies as attribute values may change.

An action within an obligation policy may result in an operation on a remote target object. This could fail due to remote system or network failure so an exception mechanism is provided for positive obligations to permit the specification of alternative actions to cater for failures which may arise in any distributed system.

High-level abstract policies can be refined into implementable policies. In order to record this hierarchy, policies automatically contain references to their parent and children policies. In addition, a cross-reference (xref) from one policy to another can be inserted manually, e.g., so that an obligation policy can indicate the authorization policies granting permission for its activities (see Section 3.1 for an example).

2.3 Is Negative Authorization Equivalent to Negative Obligation?

Both negative authorizations and obligations are needed because they are specified independently and implemented using completely different techniques. Authorizations are specified to protect target objects from unauthorized access by subjects. Subjects, therefore, cannot be trusted to interpret authorization policies so they are interpreted by trusted access control agents within the target system [57]. The implementation of authorization policies can map onto access control lists or capabilities (c.f. operating system or database access control). In commercial organizations, authorization policies are likely to be specified by a security administrator and are subject to very strict controls. Obligation policies are likely to be defined by line-managers and there may be less strict controls on modifying obligation policies. In some organizations there is an overriding default negative authorization so that all actions are forbidden unless explicitly authorized. However explicit negative authorization can be useful, e.g., to suspend a student from access to the computer system as a punishment for misbehavior. Examples of negative authorizations which are considered to be nonfunctional (security) requirements are given in [38] “reimbursements should not be revealed to secretaries with a job classification below II.”

A negative obligation policy acts a restraint on the subject in situations where it is not practical or feasible to provide a negative authorization. Negative obligations should be read as “obliged not to” or “refrain from” and can be considered as “filters” [36] to prevent permitted actions from being performed under certain circumstances. For example in policy x4 above, the test engineers must not disclose intermediate results to the analysts or developers before the tests are completed. It would not be practical to implement this as a target-based negative authorization policy as the targets do not wish to be protected and will try to get early results from the test engineers. In addition, the test engineers may actually be authorized to disclose test results to analysts and developers. Therefore, the subject must interpret the negative obligation policy and filter information going to the analysts and developers. Another example would be an agent that is authorized to perform an
action, but must not do so when in standby mode. This must be specified as a negative obligation since the internal of the agent can only be determined at the agent side. Negative obligations to refrain from actions have also been used in [33] in a similar way. In [40], the authors define a negative Deontic obligation known as a waiver, which corresponds to not obliged to whereas our negative obligation is obliged not to perform an action which is permitted.

Negative obligation policies are restraints which have to apply over long periods of time (as do authorization policies) so they cannot be triggered by events. However constraints can be used to limit their applicability.

2.4 Policy Implementation Aspects
The policy service provides tool support for defining policies and disseminating policies to the relevant agents that will interpret them. Policies are implemented as objects which can be members of domains so that authorization policies can be used to control which administrators are permitted to specify or modify policies stored in the policy service.

An overview of the approach to policy enforcement is given in Fig. 2. An administrator creates and modifies policies using a policy editor. He checks for conflicts, and if necessary modifies policies to remove the conflicts. Authorization policies are then disseminated to target security agents as specified by the target domains and obligation policies to automated manager agents as specified by the subject domains. Policies may be subsequently enabled, disabled, or removed from the agents. Manager agents register with the monitoring service to receive relevant events generated from the managed objects. On receiving an event which triggers one or more obligation policies, the agent queries the domain service to determine target objects and performs the policy actions, provided no negative obligation policies restrain the action. More details on the syntax, semantics and implementation issues of the policy service can be found in [28], [29], [30].

2.5 Roles
Organizational structure is often specified in terms of manager positions such as department manager, project manager, analyst or ward-A nurse. Specifying organizational policies for human managers in terms of manager positions rather than persons, permits the assignment of a new person to the manager position without respecting the policies referring to the duties and authorizations of that position. The tasks and responsibilities corresponding to the position are grouped into a role associated with the position (which is essentially a static concept in the organization). A role is thus the manager position, the set of authorization policies defining the rights for that position and the set of obligation policies defining the duties of that position. These definitions correspond to the concepts of classic Role Theory which postulates that individuals occupy positions inside an organization and associated with the position are a set of activities (including the required interactions) that constitute the role of that position [2]. Example roles would be a project manager or analyst in a programming team and ward nurse or staff nurse in a hospital.

Manager positions can be represented as domains and we consider a role to be the set of authorization and obligation policies (the arrows in Fig. 3) with the Manager Position Domain as subject. A person can then be assigned to or removed from the position domain without changing the policies as explained in [51].

There is a need for interactions between roles, e.g., delegating a task from a project manager role to an analyst or coordinating access to objects shared between multiple roles. The relationship between roles such as ward nurse and staff nurse is repeated in many wards within a hospital. We have, therefore, defined a Role Relationship class which can be instantiated and specifies:

- authorization and obligation policies specific to the relationship between roles,
- those policies which refer to shared target objects,
- concurrency constraints relating to policy actions for those policies specified in the relationship object,
- protocols specifying the required interactions between roles, e.g., how the project manager delegates a fault-report to be handled by an analyst.

This permits the specification of contractual relationships between roles in terms of the rights and duties of the related parties toward each other and protocols for required interactions between them. The extended role model is
described in [23], [25] and further work on roles, relationships, and our object-oriented approach to these concepts is described in [24].

Distributed systems contain a large number of objects to be managed; hence, a large number of policies are needed to cover different management functionalities, such as configuration, security, fault handling, and performance. Since conflict detection algorithms are computationally expensive, a prime concern is the choice of a scope for the conflict search. Roles, relationships, and domains offer a means of determining and progressively extending this scope. The policies within a role define the rights and duties associated with the position inside the organization. Hence, a role’s specification must be free of conflicts. The scope of the search for conflicts can then be extended horizontally to the relationships the role participates in and the related roles themselves, or vertically to all the roles which are members of the same domain or parent domains and to the policies specified in terms of parent domains which propagate to the role.

Roles have sometimes been used in Process Modeling as an abstract representation or as placeholder [10] for the stakeholders in the software development process. Roles which are assigned to individuals can then be associated with tasks and responsibilities [16]. Mapping the activities performed by roles onto a time-frame leads to the specification of Role Activity Diagrams (RAD) [3], which are also adopted in some Object-Oriented Modeling Notations [41]. A different model of RAD based on the concept of n-party interactions [11], [12] is described in [50], [43]. An extensive study of roles and role-modeling issues can be found in [25].

3 CONFLICTS

In this section, we give some example policies and indicate how modality conflicts can arise due to positive and negative policies. We also discuss conflicts arising from meta-policy specifications, i.e., constraints on the permitted policies within the policy service.

3.1 Some Examples

A service provider offers its users access to a travel-booking agent. Access is regulated according to the type of the users (private clients, corporate, etc.) and the area from which the users are accessing the service. For example:

/* Users accessing from a Network Access Point in Wales are not allowed to access the service */

p5 A- @/users_by_NAP/Wales { browse(); purchase() }
/services/Travel_book

/* All corporate users are allowed to access the service */

p6 A+ @/corporate_users { browse(); purchase() }
/services/Travel_book

An obvious conflict occurs when a corporate user accesses services from Wales, in which case access cannot be decided without giving precedence to one of the policies and ignoring the other.

Consider now the policies regulating the medication of patients in a hospital. An initial abstract policy can then be written as:

/* Nurses must maintain patients’ temperature within normal limits */

h1 O+ /* nurses */

{ /* maintain temperature within normal limits */
  /* patients ’child h2, h3, ... */
}

This policy indicates a state that must be maintained but does not specify how to do it. It is refined to a set of policies specifying the drugs which must be administered and which additional actions must be taken, plus an authorization policy to permit analgesics to be administered.

/* Administer analgesics when temperature is too high */

h2 O+ on high_temperature(patient) /nurses

{ administer(analgesics) } u:@/patients
when u==patient parent h1 , xref authorization h3

/* Nurses are authorized to administer analgesics */

h3 A+ @/nurses { administer(analgesics) }

@/patients parent h1

/* Nurses must log their handling of drugs */

h4 O+ on drugs_administered @/nurses { update() }
/drugs_db

parent h1

Note that at this point there are no authorizations giving nurses access to the database. If the administrator omits to specify such a policy, wrongly assuming it may have been specified in the general access control policies, the unauthorized obligations should be detected. Furthermore, if such an authorization does exist, it may conflict with the following policy:

/* Every night at 1:00 a.m., drug stock-levels must be checked then new drugs ordered */

h5 O+ at 01:00/@agents/stock_taker

{ check_stock(); generate_order() } /drugs_db

This conflict is due to the fact that no updates can be performed on the database while the stock levels are being checked.

3.2 Conflict Classification

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 as shown in Fig. 4, and so can be determined by syntactic analysis of polices. There are three types of modality conflicts:

1. $O^+/O^-$ the subjects are both required and required not to perform the same actions on the target objects.
2. $A^+/A^-$ the subjects are both authorized and forbidden to perform the actions on the target objects.
3. $O^+/A^-$ the subjects are required but forbidden to perform the actions on the target objects (obligation does not imply authorization in our case).

As mentioned in Section 2.2, $O^-/A^+$ is not a conflict, but may occur when managers must refrain from performing certain actions as specified by a negative obligation.

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 organization. An example of this type of conflict arises from the principle of separation of duties [4], e.g., the same manager cannot authorize payments and sign the payment checks. These conflicts are application specific and cannot be determined directly from the policy specifications—additional information is needed to specify the conditions which result in conflict. These can be specified as a Meta-policy, which is a constraint about permitted policies. (The constraints on the permitted policies within a system may be considered a policy decision—hence the term “meta-policy”). Several types of application-specific conflicts such as: conflict of priorities for resources, conflict of duties, conflict of interests, multi-managers conflict and self-management conflict have been identified in [37] and classified according to the overlaps between the subject, action and target sets. These will be described further in Section 5.

Modalities conflicts arise from overlapping domains but it is impractical to prevent these overlaps (see Section 4.1) as there is a need for multiple policies to apply to a domain to reflect partitioned responsibility and the diversity of management functions that can be performed on target objects, e.g., different managers may be responsible for maintenance and security relating to a domain of workstations. In the following, we discuss the precedence relationships which can help to resolve modality conflicts, then describe our approach to specifying meta-policies to detect application specific conflicts.

4 MODALITY CONFLICT DETECTION AND RESOLUTION

Conflict detection between management policies can be performed statically for a set of policies in a policy server as part of the policy specification process or at run-time [49], [32]. The specification time conflict detection is analogous to compile-time type checking for programming languages in that it reduces run-time errors and detects specification errors. The limitation of static analysis is that it may not be possible to evaluate policy constraints, as they depend on run-time state, and domain membership may change at run-time, so only potential rather than actual conflicts can be detected. Both static and run-time conflict detection are needed, but this paper concentrates on a static conflict detection tool which assists the users specifying policies, roles and relationships. In the following, we discuss some principles for the detection of the modality conflicts and present an implementation of the conflict detection tool.

4.1 Modality Conflicts

The analysis for modality conflicts of a set of policies enumerates all subject, action, target tuples which have a different set of policies applying to them. If there are two or more policies applying to a tuple then there is a potential conflict and the policies can be checked to see whether there is an actual conflict, i.e., a positive and negative policy with the same subjects, targets and actions.

Consider the policies $P_1$ and $P_2$ represented in Fig. 4 with $P_1$ being positive and $P_2$ being negative. Let us call the overlapping areas $a_1, a_2,$ and $a_3$ for common subjects, actions and targets. The triple overlap between the policies $P_1$ and $P_2$ creates three tuples to which different sets of policies apply:

- $P_1$ alone applies to $<s_1 - s_2, a_1 >$
- $P_2$ alone applies to $<s_1 - s_2, a_2 >$
- $P_1$ and $P_2$ together apply to $<s_1 - s_2, a_3 >$

As $P_1$ is positive and $P_2$ is negative, a conflict will be indicated. The above analysis is purely syntactic and requires no understanding of the policies. Detecting these modality conflicts is not particularly difficult. It is more interesting to consider whether the conflicts can be automatically resolved by assigning precedence to policies.

4.2 Policy Precedence Relationships

As previously mentioned, modality conflicts result from a triple overlap between the subjects, actions, and targets of the policies. In a typical organization there will be some general policies pertaining to all staff in the organization as well as more specific policies relating to staff in a department or section. Staff may also be members of many different domains. Detecting the triple overlaps between policies with modalities of opposite signs would, therefore, detect many potential conflicts that do not result in actual conflicts. Consider for example the following policies:

\[
/ \text{All users are forbidden to reboot workstations}\ /
\]

\[
W1 \text{- @} / \text{users \{ reboot() \}@/workstations}
\]

\[
/ \text{The system administrators are authorized to reboot the}\ /
\]

\[
/ \text{workstations}\ /
\]
W2 A+ @/users/sys_admin { reboot() } @/workstations

To resolve this conflict, it is necessary either to change policy W1 or to exclude the system administrators from the /users domain. Changing a policy is a lengthy operation, which requires retraction of the policy from all the agents, editing it and redistributing the new policy to all the agents. Furthermore, authorization to reboot a particular workstation may also be granted to a student engaged in an operating systems project, or testers of new hardware configurations. So, rewriting a policy may not be convenient or desirable in the general case. Removing the system administrators from the users domain is not a desirable alternative either, since this means withdrawing them from all the other policies specified in terms of the users domain. We must, therefore, allow the two policies to coexist within the system and determine which policies should apply for each manager (or set of managers), and which policies should be ignored (e.g., W1 for system administrators in the case above). Using a policy precedence relationship can substantially reduce the number of conflicts between policies and permit apparently inconsistent specifications. There are several principles, outlined below, for establishing this precedence. The choice between them has to be guided by which conflicts should be ignored and how easy it is for the human user to understand the decisions and selection of the conflict detection tool using this principle, i.e., how intuitive the principle is.

4.2.1 Negative Policies Always Have Priority

It is quite common for negative authorization policies to always override positive ones so that a forbidden action will never be permitted. However, in the example above, this implies that Policy W1, being negative, has priority over W2 so the system administrators are denied access to the system files but then they cannot perform their function. Precedence based on modality, i.e., negative policies take precedence over positive ones or vice versa, allows conflicting policies to coexist but resolves all the conflicts in a deterministic way which is not flexible. For example, the following policy may be added to the policies W1 and W2 above:

" Junior employees are not allowed to reboot workstations providing persistent services "/

W3 A- @/employees/junior_employees { reboot() } @/workstations/persistent_service

A user can be at the same time a junior employee and a system administrator. So, if positive policies override negative ones then junior system administrators will be allowed to reboot all workstations according to W2. If on the other hand, negative policies take precedence over positive ones then none of the system administrators can reboot workstations according to W1. Some flexibility may be introduced by adopting a default policy such as: everything is implicitly forbidden, or everything is implicitly authorized and defining precedence between explicit authorization, explicit denial, implicit authorization or implicit denial [34]. A default negative authorization policy, would mean that Policy W1 above would not need to be specified and so would eliminate some conflicts but does not really solve the problem. The same situation may arise for subdomains of the sys_admin domain—network administrators are not allowed to reboot workstations but a subset of them must be able to reboot workstations providing networking services such as DNS.

4.2.2 Assigning Explicit Priorities

A user can assign explicit priority values to policies to define a precedence ordering, but meaningful priorities are notoriously difficult for users to assign and may result in arbitrary priorities which do not really relate to the importance of the policies. Inconsistent priorities could easily arise in a distributed system with several people responsible for specifying policies and assigning priorities.

4.2.3 Distance between a Policy and the Managed Objects

The concept of calculating the distance between a rule (policy) and the objects it refers to has been introduced in [22] for authorization policies in an object-oriented database. Priority is given to the policy applying to the closer class in the inheritance hierarchy when evaluating access to an object referenced in a query. Consider a foreign student class to be a subclass of student, which is a subclass of person. Then an access policy applying to a foreign student overrides the general access policy applying to a person. The distance between the policy and the (class of) objects to which it applies indicates the relevance of the policy to those objects. An organization may define new policies which are intended to replace older ones so more recent policies may take precedence in some cases. In general there is a compromise between the complexity and the intuitiveness of the distance to be evaluated. A distance that is intuitive may not correctly evaluate the importance of a policy in all the cases. However, a complex calculated distance may not be intuitive enough for the human user to understand the selection and priorities assigned to a policy during the conflict detection process. For example, the priority could be based on a function of the refinement level of the policy, last modification date and author of the policy.

Spanoudakis [53] uses different types of distances to detect similarities and potential discrepancies between requirements specifications. Three distance functions are considered: 1) a classification distance giving an estimate of the analogy of two objects by measuring the importance of their noncommon classes in a generalization hierarchy, 2) a generalization distance which also takes into account the object’s superclasses, and 3) an attribution distance evaluating the similarity of analogous and unique attributes of objects.

A precedence relation similar to the one used in [22] is also encountered in the area of default reasoning. In Modal Action Logic (MAL) a default is a statement which is true unless some stronger sentence overrides it. Structuring a MAL specification in terms of objects related in a generalization hierarchy allows a specificity principle to be defined which gives priority to defaults about a specific class of objects over those for a more general class [45].
4.2.4 Specificity Related to Domain Nesting

The principle here is that a more specific policy, i.e., a policy applying to a subdomain refers to fewer objects so overrides more general policies applying to an ancestor domain. This concept has been introduced in Miró [17] and is a particular case of the previous concept of distance. Considering the specificity of a policy with regards to the objects it applies to is an intuitive concept in a domain-based system. A subdomain of objects is created for a specific management purpose—to specify a policy that differs from those applying to the objects in the parent domain. The system administrators in Policy W2 above are a subdomain of users so W2 has precedence over W1 which prohibits users from having access to system files, but other policies applying to all the users still apply to the system administrators. Similarly, a policy specified with regards to a subset of target objects, such as workstations maintaining persistent services, should take precedence over policies relating to workstations in general. Precedence based on domain nesting can thus be used to allow conflicting specifications by automatically resolving some conflicts.

The specificity precedence as used in [22] and in default reasoning [45] is only between those objects which belong to the same generalization hierarchy. Similarly, precedence based on the specificity in an I&A hierarchy has also been used in knowledge representation systems based on semantic networks and frames such as the KEE system [20], [18]. However, our precedence is based on domain nesting indicating PartOf relationships where the domains may contain the same or different types of objects. This is very flexible as objects can be grouped into subdomains to reflect specialization or any other relationship which is considered important for management purposes e.g., geographical partitioning, organizational or network structure.

In Section 4.3 we describe how domain nesting can be used within conflict detection to reduce the number of potential conflicts. We recognize that this principle does not apply successfully to all the situations, i.e., there are cases in which it is desirable that a global policy overrides more specific ones. For this purpose, the conflict detection can be performed with precedence relationships optionally disabled. The following two sections examine the importance of the overlaps between domains while applying the domain nesting principle and indicates the cases where inconsistencies still remain.

4.3 Resolving Conflicts Based on Domain Nesting

In Fig. 4, P1 and P2 have opposite modalities and neither is more specific so a conflict is indicated to the user. Now consider a policy P3 (shown in Fig. 5) defined by the tuple < s1, a3, t3 > such that s3 = s1, a3 = a1, and t3 is a subdomain of t2 which is a subdomain of t1.

We now have the following tuples and policies:

- P1 alone applies to < s1 - s1, a1 - a1, t1 - t1 >
- P2 alone applies to < s2 - s2, a2 - a2, t2 - t2 >
- P1, P2, and P3 together apply to < s1, a1, t3 - t3 >
- P2 and P3 together apply to < s1, a2, t3 - t3 >

P3 is positive and is more specific than P2 so it overrides P2 in the areas where they overlap, i.e., for the tuple < s1, a3, t3 > and < s1, a3, t2 >. Since P1 and P3 have the same modality, the conflicts can be automatically resolved using domain nesting precedence.

Note that when displaying the result of a conflict detection check it is important to provide the user with the information regarding which policies conflict, where precedence resolves conflicts and to which tuples < subjects, actions, targets > these policies apply.

If policies were specified in a logical formalism, the use of domain nesting precedence would require nonmonotonic capabilities of the logical framework. For example, consider that P1, P2, and P3 above are authorizations. Before policy P3 is added, managers in the s3 domain are forbidden to perform the invocations denoted by a3 on all the targets defined by the t3 - t1 set. This is because only P2 specifies their access rights on those target objects. When policy P3 is added, the managers are now authorized to perform the invocations on those target objects since P3 is a positive policy and overrides P2.

4.4 Limitations of Domain Nesting Based Precedence

The domain nesting precedence determines all policies which apply to a tuple of subjects actions and targets and gives precedence to policies which apply to a more specific set of subjects, targets, or both. There are cases in which precedence cannot be established because the sets are equal, the subject sets are more specific but the target sets are less specific or vice versa. Various situations where precedence can or cannot be established between two policies are illustrated in Fig. 6. Note that that precedence may based on a policy's subject or target set so it is not an ordering relation because it is not transitive. However, it presents the advantage of catering for both more specific subjects and more specific targets. There is no precedence relationship between obligations and authorizations since an obligation overriding an authorization would convey the implicit assumption that the obligation implies authorization and this is not true for our policies.

5 Meta-Policies

Modality conflicts can be detected purely by syntactic analysis of the policies. Application-specific conflicts arise from the semantics of the policy and are specified in terms of constraints on attribute values of permitted policies. For example, in the case of the separation of duties [4] where the same set of managers are not allowed to authorize payments and sign the payment checks, the conflict is specific to the actions specified, i.e., authorize a payment
and sign the check. It, therefore, must be specified by an additional constraint, which when evaluated, detects the conflict. We term these constraints meta-policies, i.e., policies about management policies, which requires the use of quantifiers over sets of policies.

For example, the separation of duties can be stated as "there should not be two policies having overlapping subject domains which give rights to authorize a payment and sign a payment check." This can be written as a logical predicate:

\[
\forall P_1, P_2 \in \text{/policies/accounting} \\
\text{intersectSubject}(P_1, P_2) \land (\text{authorize} \in P_1.\text{actions}) \\
\land (\text{sign} \in P_2.\text{actions}) \land (\text{payment} \in P_1.\text{targets}) \\
\land (\text{cheque} \in P_2.\text{targets}) \land (P_1.\text{mode} = 2, \text{mode} = \text{A+}) \\
\implies P_1 \text{conflicts with} P_2
\]

It would not be practical to specify the above meta-policy as constraints within authorization and obligation policies as these are evaluated every time the authorization policy is checked or the obligation policy is triggered. Evaluating this type of constraint would require checking the policy service to determine whether another policy exists which violates this constraint. The run-time overheads of this would be prohibitive. Specifying the constraint as a meta-policy permits it to be evaluated once at specification time, when new policies are added. In addition, there is a conceptual difference in that the obligation and authorization policy constraints limit the applicability of these policies whereas meta-policies are constraints about permitted policies in the policy service.

We have been experimenting with meta-policies by implementing the predicate specification in Prolog. The set of policies contained in the Conflict Detection window (Fig. 7) is automatically translated into Prolog assertions and the predicates are evaluated. The set of all the solutions to a predicate is the set of policies that are in conflict. Several types of application specific conflicts are presented in [37]. They identify cases such as the conflict for resources, multiple management and self-management that are briefly summarized here.

5.1 Conflict of Resources

This occurs when the amount of resources (target objects) available is limited. The policies obliging and authorizing managers to use these resources must, therefore, have a limited number of objects in their target scope. For example "at most five disk partitions can be used for backup activities."

5.2 Multiple Management

Multiple managers may manage the same objects, either because the objects are shared between several tasks or
because different management functions are assigned to different roles. This may constitute a conflict when the management operations to be performed on the target object are not independent. For example, an update operation may require a service to be temporarily shut down, while a get_configuration operation may require it to be in service.

5.3 Self-Management
A manager may not be allowed to retract policies that he is supposed to perform. This can be written as: “There should be no policy authorizing a manager to retract policies of which he is the subject.”

Although in [37] these conflicts are characterized by their overlaps between subject, action and target sets, no assumption can be made in the general case. The separation of duty conflict may not have an overlap between target domains. Further, it may not even have an overlap between the subjects domains if managers in the same accountancy department are not allowed to both authorize payments and sign checks.

Further work remains to be done regarding the specification of meta-policies. While the use of Prolog is attractive because it offers the flexibility of a general logic programming language a more restricted notation relating directly to the attributes of a policy is desirable.

6 Tool Support
The prototype conflict detection tool currently detects overlaps between policies and optionally applies domain nesting based precedence. The domains and policies are distributed among several servers so CORBA remote object invocations [39] are used for retrieving the policies and querying domains to evaluate their sets of subjects, actions, and targets. In theory, all policies in the system need to be checked for overlaps, but this is impractical. Instead, we permit the user to specify the scope of policies to be checked, for example, the policies applying to particular roles or the policies of a relationship between roles. Policies or domains of policies can be dragged from a domain browser tool into the conflict detection window to establish the set of policies over which the conflicts are to be detected. The meta-policies discussed in Section 5 can also explicitly define the scope to which they apply.

Since there are cases in which a more specific policy should not take precedence, domain-nesting precedence can be optionally disabled so that all the policies which potentially conflict are indicated. When enabled, the precedence relationship between policies is indicated by arrows between the policy icons, as shown in Fig. 7, so the user can easily determine which policies override. Finally an analysis option also permits all the tuples of subjects, actions and targets and the policies applying to them to be displayed even if there are no conflicts as it is useful to examine which policies apply to which tuples.

6.1 Example
Often software process management systems use objects such as modification requests (MR) or trouble-tickets in order to coordinate the actions of the actors in the development process. However, these systems can rarely reflect the complexity of the organizations, which may share developers between teams or develop modules common to multiple projects. We describe a simplified example relating to the management of modification request objects in order to highlight how conflicts may arise in a policy-based specification and how they can be detected using the analysis tool developed.

Modification request objects are created by invoking the create_MRI() method on the MR_factory object. The organization is divided into two project teams, each headed by a project manager (Fig. 8). A group of engineers developing network modules for multimedia streams (streamingAPI) are shared between the two projects. In project1 this group is a subgroup of a larger group of network developers (NWdevelopers). We use domains to represent the grouping of objects and implement policies as objects also represented in the domain hierarchy (Fig. 8). Project2 has a manager appointed to the help desk in order to deal with problems encountered by customers. Note that in Fig. 8, the
Fig. 8. Software development environment domain structure.

shaded boxes correspond to nondomain objects, i.e., policies and managers.

Policies are used to specify who in the organization, is permitted or forbidden to create modification requests. By default, we will assume the environment to be an open one in the sense that all invocations are authorized unless explicitly forbidden. Access control regarding the creation of modification requests is decided within each project separately, based on the characteristics of the project. Let us consider that both project1 and project2 adopt a default policy where only the project manager is allowed to create MRs. This means that the members subgroup of each project is prohibited from creating MRs by two policies specified as below.

p1 A- @/organization/project1/members { create_MR() } /MRFactory
p3 A- @/organization/project2/members { create_MR() } /MRFactory

Furthermore, any modifications to the network connection modules may give rise to changes in the entire project1. Therefore, network developers in project1 are authorized to create modification requests for the whole project.

p2 A+ @/organization/project1/members/NWdevelopers { create_MR() } /MRFactory
Policy p2 should be in conflict with policy p1 since members of the NWdevelopers group are subjects of both p1 and p2, which have opposite modalities. However, since policy p2 is more specific than p1 (it relates to a subdomain of project1 members), it will take precedence over p1 according to the domain nesting principle. This will be detected by the conflict detection tool and indicated by an arrow between the two policies (Fig. 7).

The help desk engineer in project2 receives direct feedback from the users regarding possible errors or misbehaviors of the product. An obligation policy ensures that any error reports will be transformed into modification requests that will be investigated by the developers (policy p4).

p4 O+ on error_reported @/organization/project2
members/helpdesk { create_MR() } /MRFactory

Fig. 7 illustrates the main conflict detection window. The administrator chooses the set of policies on which the conflict detection is to be performed (here p1-p4) via a drag-and-drop mechanism. After each conflict check, precedence relationships between policies will be indicated to the administrator if precedence has been enabled. Allowing the administrator to detect conflicts without using precedence relationships is necessary because there are cases in which general policies should not be overridden by more specific ones such as: Remote access to classified information is to be denied to all users regardless of the circumstances.

After checking for conflicts with precedence enabled, two conflicts have been found in the policies p1-p4. These conflicts may be displayed in windows as shown in Fig. 8. The first conflict (a) outlines that analyst1 and developer1 are subjects of two conflicting policies among which no precedence exists. In the domain structure this corresponds to the members of the streamingAPI subgroup, which is shared between the two projects. In project1 this subgroup is authorized to create modification requests by virtue of its being a subgroup of network developers but is prevented from doing so by the default policy specified in project2 for all the project members. The second conflict (b) shows that the helpdesk staff in project2 is subject to two policies giving rise to an O+/A- conflict. Policy p4 obliges the
manager to create a MR when customers have reported an error. However, he is prevented from doing so by the default policy that forbids MR creation to all members in project2. Nesting of the subject domain does not help because our obligation policies do not imply authorization, as mentioned in Section 4.4.

The conflict detection tool is integrated with other tools for policy editing and domain browsing. The system administrator or policy maker can, therefore, study the indicated conflicts, as shown in Fig. 9, and then edit the policies or query domain membership of subjects and targets.

This example has been greatly simplified by considering a single target object, a single action to be invoked on this object and a limited number of subject domains and policies. In any realistic situation, where there are many levels of nested domains, the likelihood of specifying conflicting policies with overlapping domains is far greater. Furthermore, several administrators may be responsible for specifying policies and modifying the domain structure thus increasing the risk of conflicts in the specification. In addition to detecting conflicts, our tools can be used to analyze a domain specification to determine which policies apply to which tuples of subjects, actions and targets.

7 RELATED WORK

Our concept of domain nesting precedence is based on that of Miró [17], but they only deal with authorization policy for file system security. We have discussed in Section 4.2 the relationship of our work to that of [22] and the specificity of default reasoning [45] and knowledge representation systems based on semantic networks and frames [20]. Sandhu [46] presents constraints that are similar to our meta-policies, but the notation used and the enforcement of the constraints are not described. Minsky’s "law governed systems." [35] can also specify permissions and prohibition as a set of rules which are similar to our positive and negative authorizations. Conflicts are avoided by defining a meta-level rule which specifies whether a permission or prohibition take precedence and override the other. Meta-level rules can also be used to define a "Prohibition-based regime" where permission is the default unless explicitly prohibited, or a "Permission-based regime" in which prohibition is the default unless explicitly permitted. As discussed in Section 4.1, we found assigning precedence to positive or (more commonly) negative policies very limiting and not as intuitive as precedence based on specificity. However, we have also identified the need to be able to flexibly specify the policy precedence relationship, used by the analysis tools, in terms of a meta-policy, although this is not yet implemented in the current version. We actually only implement positive authorization and remove negative authorizations by refinement of the policies. We also assume that all actions are prohibited unless explicitly authorized. However there are systems which do implement negative authorization, particularly for database access control so we consider it necessary to be supported by a general purpose policy specification toolset. The system described in [35] is not distributed although in [34] a common global set of constraints is implemented by means of filters in every node which check that all interactions are consistent with the global law. We assume both specification and implementation of policies can be distributed. Our policies are not global but are interpreted only by explicitly specified subjects (for obligations) and targets (for authorizations).

Sibley et al. have also identified the need for a Policy Workbench with automated tools to specify and analyze policies [47]. They have experimented with both first order logic and an object-oriented approach to representing policy. They found that the latter reduces the size of the policy base and simplifies policy specification [48]. The policies considered are not limited to obligations and
authorizations but general rules about the system. Both policies and "real-world" facts are eventually formalized in first order predicate calculus with the use of modeling techniques such as enhanced entity-relationships diagrams. A theorem prover is then used on the policies and real-world facts to ensure consistency of the specification [31]. Inconsistencies can be of two kinds: 1) logical inconsistencies and 2) statements that can be proved by the theorem prover but do not comply with the intended specification. The tool is, therefore, used in a first step to detect logical inconsistencies in the policies and real-world facts given as axioms. Questions can then be asked by the policy maker in form of theorems to be proved by the tool. The authors have also explored a third stage performing "what-if" analysis by querying the theorem prover with regards to incremental changes in the policy base. The complexity of the approach of Sibblesy, Michael, and coauthors is due to the generality of their policies. Our policies are simpler and explicitly identify subjects, actions and target objects. A logic-based approach, where real-world facts including object state need to be modeled is more suited to requirements and specification analysis than distributed systems where objects states are often transient in nature. Nevertheless, their approach would help with respect to our policy constraints, but they do not use precedence relationships between policies to automatically resolve some conflicts. By treating basic conflicts between policies in terms of logical inconsistencies, implementing precedence into the conflict detection process would require either prioritized or retractable goals or more complex formalization of the policies catering for logical exceptions. Consider the case of a group of managers being authorized to perform an action by policy \( P_X \) that is forbidden by a more general policy \( P_Y \). Implementing precedence would require either: 1) retracting the logical facts corresponding to the prohibition given by \( P_Y \) and replacing those facts by their negation, i.e., the authorization given by \( P_X \), or 2) formalizing policy \( P_Y \) in a different manner—managers (with specified exceptions) are not allowed to perform actions on targets. Policy \( P_X \) would then need to be explicitly declared as an exception to \( P_Y \).

Prakken addresses these issues and shows that defeasible or nonmonotonic reasoning is required when formalizing exceptions to the rules in a logical system [42]. Although a finite number of exceptions can be formalized in first order logic, a law (and also a policy) can have an infinite number of exceptions which cannot be foreseen or taken into account when specifying the policy in the first place. Prakken conducts an extensive review of the defeasible logic frameworks suitable for dealing with exceptions and of the various precedence relationships or collision rules that can be established between the formalized rules specified in the logical system. Among other precedence relations, the specificity principle (domain nesting in our case) holds a prime place for choosing between conflicting conclusions. Prakken then extends the argument by modeling both defeasible reasoning and reasoning with inconsistencies between rules as comparing the arguments for incompatible conclusions. This comparison is carried out by defeating the applicability of the rules invoked in the

argument using precedence relations. The logical framework used is complex, the author himself expressing concerns about the feasibility of its implementation as an automated tool. Our policy specification is much simpler and considers precedence relationships. Specifying activities and objects using first order logic within a distributed system would encounter additional problems relating to failures and transient nature of some objects within the system. Other work on combining precedence relations can be found in [1].

Another approach, used to detect feature interaction in telecommunication systems [14], [56], considers policies as goals and uses a hierarchical representation of goals and alternative ways to achieve those goals. Agents that implement policies negotiate with each other in order to find a plan in the goal hierarchy that achieves the goals of all the agents but does not involve conflicting activities. In this work, conflicts are also considered as logical inconsistencies. The originality of the work is the use of negotiation for achieving conflict resolution. Planning techniques for conflict management are also used in Distributed Artificial Intelligence [21]. In the case of our management policies such techniques could be used only in conjunction with the refinement of the policies. For example, an abstract policy may be implemented in different manners by alternative sets of more concrete policies. Koch [19] uses a policy notation based on ours and establishes a semantic graph model to detect ill-behaved policy sets with unsatisfiable preconditions. This can also be used to perform "what-if" analysis on chains of policies prior to execution.

Deontic Logic provides the closest approximation of our management policies in the context of a logic system. However as described in [15], Standard Deontic Logic also relies on the axiom of indefinability which defines a permission as \( P_X = \text{Def. } \neg \text{O} \rightarrow \neg \text{x} \). No such assumption is made between our authorization and obligation policies. A number of new logical systems with slightly different axioms are emerging but the struggle against the paradoxes that can be proved in such systems seems to continue. Ong [40] also detects conflicts between positive and negative deontic obligations and permissions but treats conflicts only as logical inconsistencies.

In the context of Requirements Engineering, Dubois [8], [9] uses the deontic constructs of obligation, prohibition and exclusive obligation to define constraints over the potentially infinite set of behaviors of an agent. Their default policy is an open one, i.e., all behaviors which are not prohibited are by default permitted. Although these deontic constructs may appear similar to the policies described in this paper there are substantial differences. The underlying notion of deontic permission (prohibition) should not be equated to a positive (negative) authorization policy. Permissions and prohibitions are used in [9] as constraints on the possible behaviors of the agent. In our framework, positive and negative authorizations are statements about the behavior of the access control system which directly interprets and implements them. In this respect, their deontic prohibitions are closer to our negative obligation policies. On the other hand, their obligations are essentially
static, i.e., something is obliged under given conditions, while our positive obligations are event triggered and thus closer to their causality constraints. Note, however, that the constraints defined in their work are local to each agent and, therefore, cannot overlap in their scope. Hence, conflict detection is reduced to the problem of logical inconsistency, and since no specialization relationship exists between agents, no precedence or specificity is considered.

Considerable work regarding conflict detection and resolution and goal (re-)structuring stems from the requirements engineering community [38], [55], [44]. While these concepts are relevant for dealing with policy refinement (see Section 8 below), there are significant differences between the end products of the two processes. Requirements engineering techniques essentially develop a model of the system which renders explicit "what" the behavior of the system must be, "why" the system is needed and should behave in the prescribed manner and eventually "how" the system is constructed [13]. The end product of the requirements phase is a requirements specification document defining the system characteristics which need to be implemented during the design phase [9]. Therefore, restructuring subgoals in order to resolve conflicts carries no overheads in the run-time system. In our case, the end product of the policy refinement process is a set of policies which are directly interpreted by the agent agents and access control systems. If the introduction of a new policy results in a conflict, changing existing policies requires removing them from the (potentially) distributed agents and replacing them with new ones. Because of agent distribution, the time required for such operations may be lengthy and conflict resolution techniques should seek to minimize the number of policies to be modified. Note, that by restructuring and conditionalizing [44] all modality conflicts can be removed since it is always possible to specify policies applying to nonoverlapping domains. However, as mentioned above and discussed in Section 4.2, this is not always desirable in our case although sometimes it cannot be avoided. The relationships between conflicts as described by Robinson [44] (e.g., does removing conflict Ci also remove conflict Cj) are particularly interesting when applied in the context of the domain hierarchy and would require further investigation. In particular, it should be possible to automatically determine such relationships between modality conflicts from the policy specification and the domain structure.

The work presented in [6] will form the basis for our future work to apply requirements engineering techniques to the policy refinement. In particular the handling of the high-level policies as goals (nonfunctional requirements) and their operationalization into concrete policies which influence the behavior of the system being managed could be applied in our case. However, this requires further study (see Section 8 below) since the specification of interpreted policies does not have a straightforward expression. For example, triggers in the Performs relationship [6] are modeled as conditions while our positive obligation policies are triggered by event notifications emitted by the monitoring system.

8 Conclusions and Future Work

The policies described in this paper are interpreted so can be dynamically replaced or enabled to change the management strategy within a distributed system. There may be multiple administrators specifying and modifying policies which can lead to conflicts between the policies. The paper has presented the integration of a conflict detection tool in a role and policy-based framework. We perform off-line, static analysis of a set of policies to determine two types of conflicts: 1) modality conflicts, arising from positive and negative policies, which can be checked by analyzing the syntax of the policies and 2) application specific conflicts that need to be specified by external constraints which we express as meta-policies. Modality conflicts arise from a triple overlap between the subjects, actions and targets of the policies, but it is neither practical nor desirable to prevent these overlaps. We make use of a precedence relationship based on the specificity of the policies with respect to domain nesting to reduce the number of potential overlaps indicated to a user and allow inconsistencies between policies to exist within the system, as we consider this to be an effective and intuitive precedence relationship. Roles are an important management concept but also provide a scope to limit the set of policies to be analyzed.

Another aspect of policy analysis relates to determining the policies applying to a particular subject or target. Our policies explicitly identify both subject and target and the domain service maintains the list of policies applying to a domain so this is comparatively easy to do.

We have implemented a prototype role framework which supports distributed policy and domain servers and analysis of a set of policies, indicating conflicts as well as precedence relationships. This will enable us to experiment in realistic situations and evaluate the use of the precedence relationship. Our approach is to detect as many conflicts as possible at specification time, rather than leaving them to be detected at run-time. The user can then modify the policies to remove conflicts. This has been implemented using a CORBA based distributed programming environment [39].

The paper has concentrated on static analysis of policies, but there is also a need for dynamic run-time conflict detection which is an area we are currently working on. The need for dynamic analysis is that domain membership may change dynamically and some constraints can only be evaluated at run-time as they may depend on object states or current time. Conceptually there is no distinction between static and dynamic conflict detection, but the problem is to avoid the overheads of a potentially complex analysis every time an obligation is triggered or an authorization checked. We are experimenting with a conflict agent which resides with a manager agent and maintains information on all enabled obligations and authorizations pertaining to that agent. We are trying to pre-compute as much information as possible for the policies to minimize the run-time costs. Sibley [49] and Michael [32] also discuss the relationship between static vs. run-time checking of policies. A problem arises from those conflicts which cannot be resolved automatically by some form of precedence. Passing the conflicting policies to an
administrator to resolve, as with static analysis, may not be practical with some automated management systems, because of performance constraints. We hope to detect these cases as potential conflicts by static analysis and define suitable precedence meta-policies to resolve the conflicts when they actually occur. We need to evaluate additional case studies to see whether this is practical.

Although precedence based on domain nesting works for some cases, it does not cover all situations. Sometimes there is a need for negative policies to have precedence in order to quickly withdraw services from an individual or a group. In general there is a need for more flexible application specific precedence relationships, possibly specified as a meta-policy. Our meta-policy specification language also needs further refinement. However, since meta-policies express constraints on the policies which can be specified, the changes to the notation must also be supported by further investigation of the policy refinement process.

Our policies differ from a requirements specification in that they are directly implementable so performance issues are more important in a policy notation compared with a requirements notation. In spite of this, there does not appear to be any fundamental difference between the process of refinement of high-level abstract policies into implementable ones, and the refinement of goals into detailed requirements specifications, as supported in the work of [6], [54], [38]. Our new project (see http://www-dse.doc.ic.ac.uk/projects/secpol/SecPolOverview.html) will investigate the applicability of the Requirements Engineering approach, the KAOS method and associated GRAIL environment [7] to refinement and consistency analysis of our policies. This will permit checking whether policies satisfy goals or if there are mistakes in the refinement process. Our policies currently maintain only primitive dependency relationships between them. The models of Strategic Dependency and Strategic Rationale described in [58], [59] will prove useful to represent the intentional and means-ends relationships. Note that many conflicts between management policies arise from policy overlap due to the various different functions (e.g., configuration, security management, fault handling, performance management, monitoring) which have to be performed by a management system. Clearly, an adequate representation of the organizational framework is necessary for specifying nonfunctional requirements and dependency models. Our role-based framework [25] can be used to this end since in addition to policy specification it also caters for structured conversational interaction (e.g., negotiation) between the management roles.

ACKNOWLEDGMENTS

The authors gratefully acknowledge financial support from Fujitsu Laboratories and British Telecom. Also, the authors acknowledge contributions from their colleagues to the concepts described in this paper—particularly Nicholas Yiallalis and Damian Marriott. Additionally, the authors thank the referees for their useful comments and suggestions that improved this paper. A preliminary version of this paper was published under the title "Conflict Analysis for Management Polices," Proceedings of the Fifth IEEE/IFIP International Symposium on Integrated Network Management, San Diego, May 1997.

REFERENCES

Emil C. Lupu obtained his Diplôme d'Ingénieur from the École Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble (ENSIMAG), France, in 1994 and his PhD degree in computer science from Imperial College, London, in 1998. Dr. Lupu is currently a research associate in the Department of Computing at Imperial College. His research interests include network and distributed systems management, design, security, multimedia, and mobility issues in large distributed systems. Dr. Lupu serves on numerous conference program committees, including the IFIP/IEEE Symposium on Integrated Network Management (IM'99), the IFIP/IEEE Network Operations and Management Symposium (NOMS'2000), the IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM'99), and the ACM Workshop on Role-Based Access Control (RBAC'98, RBAC'99). He is a member of the IEEE and a member of the IEEE Computer Society.

Monita Sioman obtained his BSc (Eng) degree in electronic engineering from the University of Cape Town, South Africa and a PhD degree in computing from the University of Essex, United Kingdom. Dr. Sioman has been with the Department of Computing at Imperial College since 1975. He has managed many research projects funded by the United Kingdom Engineering and Physical Science Research Council (EPSRC), European Union, and various other industries on management, security, and design of distributed systems, multimedia systems, and mobility. He is editor of a reference book on Management of Network and Distributed Systems (Addison-Wesley), coeditor of the IEE/IOP/BCS Distributed Systems Engineering Journal, and a member of the editorial board for the Journal of Network and Systems Management. He was program cochair of the First IEEE Enterprise Distributed Object Computing (EDOC) workshop and is a member of the EDOC steering committee. He was program cochair of the 1999 IEEE/IFIP Integrated Management Symposium (IM'99). Dr. Sioman is chair for the EPSRC Multimedia and Network Applications Funding Programme. He is a member of the IEEE Computer Society.