Policies and Roles in Collaborative Applications

W. Keith Edwards[1]
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
+1-415-812-4405
kedwards@parc.xerox.com

ABSTRACT
Collaborative systems provide a rich but potentially chaotic environment for their users. This paper presents a system that allows users to control collaboration by enacting *policies* that serve as general guidelines to restrict and define the behavior of the system in reaction to the state of the world. Policies are described in terms of access control rights on data objects, and are assigned to groups of users called *roles*. Roles represent not only statically-defined collections of users, but also dynamic descriptions of users that are evaluated as applications are run. This run-time aspect of roles allows them to react flexibly to the dynamism inherent in collaboration. We present a specification language for describing roles and policies, as well as a number of common "real-world" policies that can be applied to collaborative settings.

KEYWORDS: computer-supported cooperative work, policies, roles, infrastructure, access control, Intermezzo.

INTRODUCTION

Collaboration and Chaos

Collaborative systems are potentially chaotic environments. Multiple users create opportunities for collaboration, rich and potentially unexpected interactions occur between users and applications—all of these contribute to the dynamic nature of collaborative software. The dynamism present in collaborative systems approaches—intentionally—the fluidity and richness of interactions among people in the physical world.

Interactions with collaborative systems are not as predictable as with single-user systems since other users are not always predictable. Put simply, the presence of other users, with their own goals and experience levels, introduces the potential for uncertainty, unpredictability, and surprise into collaborative work sessions.

Further, as the complexity of the collaborative environment increases, users may have to contend with an increasing number of distractions and demands on their attention to "keep up" with the collaborative endeavor. They must work to sustain the mechanics of the collaboration, attending to their tools and other users, in addition to the task at hand.

Examples of work devoted to maintaining collaboration rather than performing the domain-specific task include explicit forms of session management (inviting users to a session or browsing lists of sessions), creating and maintaining role membership for a collaborative editor, accepting or rejecting requests for access to data, and interacting with floor control systems. These are all "overhead" tasks in the sense that they are required only because the collaborative environment requires them; not because of any inherent need in the task itself.

Selectively Limiting Chaos

The presence of unpredictability within a large space of actions is one of the reasons that collaborative systems can present such a rich and free-form user experience. For example, think of the open-ended forms of collaboration seen in group sketching and mediaspaces. Such chaos also indicates, however, that there may be a need to "rein in" some of the "destructive" unpredictability present in collaborative systems. If a collaborative environment passes control to the user whenever ambiguity is present, users will likely be overwhelmed by the task of "baby-sitting" the environment, rather than getting work done.

In contrast, if we can provide some general guidelines to our tools and environments about how to support the dynamism in collaboration, we can relieve users of some of the burden of maintaining the collaboration; users can be free to attend to the focus of the collaboration, rather than to their tools.

Intelligence about how to respond to particular occurrences in collaborative situations is called a *policy*. The American Heritage Dictionary defines policy as "a general principle or plan that guides the actions taken by a person or group" [1]. In a collaborative environment, policies govern the particulars of how users and applications interact with one another. A policy describes a general contingency against which specific events are evaluated and handled [13]. The goal of a policy system should be two-fold:

Copyright © 1996, Association for Computing Machinery. Published in *Proceedings of ACM Conference on Computer-Supported Cooperative Work* (CSCW'96), Boston, MA. November 16-20, 1996.

1. This research was conducted by the author while at the Georgia Tech Graphics, Visualization, and Usability Center.

- *Reduce unpredictability in the system to a "manageable" level.* Allow the system to respond in expected ways to the actions of users and other applications, and support the principle of "least surprise."
- *Require less effort from users.* Move the burden of dealing with most user and application actions from the user to the system.

Additionally, the use of policies in the context of collaborative systems imposes a potentially competing goal:

- *Do not stifle interaction*. The policy system should not be so restrictive that it discourages or artificially limits the interactions between participants in a collaboration.

In essence a policy scheme is used to selectively limit the dynamism inherent in a collaborative system. The "trick" is to strike a balance with a set of policy controls that can make a rich collaborative system manageable, and yet still preserve the essential components of dynamism that are necessary for human-to-human communication and interaction.

This paper describes a system that can implement a variety of useful policies in collaborative settings, particularly in the areas of awareness and coordination. This system is unique in several regards, first in its use of *access control* to support policies. Access control proves to be a powerful tool that is capable of capturing many of the situations commonly called "policies," and we establish several examples of policies based on access control. A second innovation of this system is that it supports highly dynamic policies. The mapping of users to policies that regulate their access can not only be made statically (at application start time), but also dynamically (changing as the application runs).

The facilities described in this paper have been implemented atop the Intermezzo collaboration support environment. Intermezzo is designed to provide applications with support for coordination, rendezvous, and awareness [7].

ACCESS CONTROL AS A BASIS FOR POLICY

The notion of policy is broad, even in the non-technical usage of the word. How can we devise a formal system that allows us to capture the extremely general set of controls that users may wish to place on their environments?

At this point, we present some examples of policies to drive the presentation of the formal policy system. The policies below are defined in common, everyday language and are immediately obvious as *policies* in the non-technical sense of the word. Further, they represent common desires of users and have obvious utility in collaborative settings. If we can capture these sorts of common, easy-to-express desires in our system, then we will be close to supporting the rich forms of policy usage that we see in the workaday world.

- "I don't mind people in my workgroup knowing what I'm working on, but others..."
- "Don't let anyone bother me when I'm working on my thesis. Unless it's my advisor of course."
- "I need to share my workspace with others during demo days."
- "If anyone calls for me, tell them I'm not here unless they've called with the new budget numbers."

All of the scenarios above are examples of policies that are common in everyday use. Users want the ability to regulate access to their information and personal space, and to govern how the system will respond to events.

This paper postulates that *a wide array of policies like the ones above can be defined in terms of access control rights on data objects.* While perhaps not all policies can be captured by an access control-based model, many of them can. Of course, making such a scheme broadly applicable to policy concerns requires that we wisely choose not only the access control primitives we will support, but also the data objects to which we will be restricting access. We shall require not only an infrastructure that provides representations for objects and operations in the collaborative environment, but also conventions for how applications will access the data objects in the environment.

To effectively support policies like the ones above, we must provide a system that is (1) expressive enough to capture a range of policy considerations, (2) flexible enough to not over-constrain the collaboration, and (3) integrated with information from and about the "real world" in which collaboration occurs.

All of the examples above deal with coordination policies: the information being regulated concerns individuals and their actions. In other words, it is information useful in providing *awareness* of others. The focus of this work is supporting coordination and awareness, although many of the ideas presented here could be extended to the domain of application-specific policies and access control. Examples include regulating access to paragraphs of text in a shared editor, or managing a floor control system.

The next sections lay the groundwork for a policy system based on our three goals of expressiveness, fluidity, and contextual integration.

The Importance of Expressiveness

What work has been done to-date in policy systems for collaborative applications? Research on policy can be broadly divided into "coordination domain" policy—policy controlling between-application information, such as awareness and location information—and "application domain" policy—policy controlling application-specific data, such as access to text in a shared editor.

In the coordination domain, access control tends to be an all-or-nothing affair. A common access control metaphor is the "closed door." CaveCAT [12], DIVA [19], Montage [21], Cruiser [17], and others all provide an access control system in which users can allow or disallow access globally. It is impossible to specifically grant or refuse access to individuals or classes of individuals. These systems might be considered "target-oriented" since access is either granted or denied based solely on the target of access; the requestor is not taken into consideration. Some of these mediaspace-like systems expand incrementally on the door metaphor by providing several categories of access restrictions such as door open, door closed, door ajar, and so on. Only recently

have some coordination-oriented systems begun to look at more novel forms of policy. For example, the ability to support "low-disturbance" forms of audio/visual interaction in Smith's work [20].

In the application domain, many systems use the notion of *roles* to define access control groups. A role is a category of users within the user population of a given application; all users in a certain role inherit a set of access control rights to objects within the application. Roles are particularly common in shared editors such as Quilt [11], which includes roles for writers (who are allowed only to change their own work), readers (who are not allowed to modify the document), and commentators (who can only add "margin notes" to the document). Roles are also found in other systems, such as ConversationBuilder [10], MPCAL [9], SASSE [2], ICICLE [3], SUITE [18], and PREP [15].

Other work in application domain access control approaches the classical security systems seen in operating system research. For example, Shen and Dewan present a robust model of the access control needs of applications [18].

The access control models provided by most of the role-based systems, as well as that of Shen and Dewan, provide expressive power beyond that typically found in mediaspace-like systems. For an access control system to be able to capture the range of needs expressed by the example policies discussed earlier, it must go beyond the rudimentary boolean schemes typically found in coordination systems.

For example, policies such as our first example ("I don't mind people in my workgroup knowing what I'm working on") require systems with some expressive power. Such systems would grant access to a set of information ("what I'm working on") to a constrained set of users that is long-lived and statically-defined ("people in my workgroup"), and disallow access to others.

The Importance of Fluidity

Human activity is a highly dynamic medium, rich in subtle interaction and constant shifts in the focus, priorities, and roles of the participants. As Moran and Anderson [14] state, "fluidity is a fundamental feature of work activity, and we need to be attuned to how technologies of various kinds can play a role here." To be useful (or even usable) in such a setting, a policy system for collaborative applications must not only be able to respond to such changing interactions, but also must not artificially restrict interaction. An overly-static policy system will serve only to cripple the dynamism that is inherent and beneficial in unconstrained human interaction. Policy systems must not only allow, but also *support* the fluidity of interaction seen in "real life" human situations.

A problem with many of the policy systems used in prior work is that they may be too rigid to support effective collaboration. For example, Dewan, *et al*., in [4], point toward more flexible and fluid specification of roles as being a requirement for effective collaboration:

> *...users should be allowed to take multiple roles simultaneously. For instance, a teaching assistant should be able to simultaneously take the roles of "student" and "lecturer."...It should be possible to dynamically change collaboration rights, roles, owners, and ownership semantics. This requirement allows, for instance, a user of a code inspection tool to graduate from an "observer" to an "annotator."...access control must be performed flexibly in the ways described above...*

Dourish and Bellotti have indicated similar concerns, stating, "there seems to be justification for arguing that role-switching in CSCW systems should, therefore, not be a complex, time consuming operations which hampers [access to shared work spaces]." [5]

For an access control system to be able to capture the wide range of needs expressed by the example policies discussed earlier, it must not only provide expressiveness beyond simply "on" or "off," it must be able to support the ephemeral styles of interaction that make human-to-human collaboration so rich.

Take as an example the second of our policy scenarios ("don't let anyone bother me when I'm working on my thesis, unless it's my advisor"). A "classical" system would be able to identify "advisor" as a privileged user who should be allowed special access. But traditional schemes typically support group membership that is changed infrequently if at all, and would not be able to respond to the instantaneous condition on which the policy depends: whether the user is working on the thesis. Such a system would require the user to explicitly modify the access control settings as the pattern of work changes.

The Importance of Context

By examining the policy scenarios outlined earlier, we can see that many share one trait in common: they are based on information about the context of the user and the task in both the "real" world and the computer-mediated virtual world. This should not be surprising: all work, even computer-based work, takes place in a physical workplace with its own demands on our time and attention. Collaboration in particular, with its reliance on "out of band" communication channels (face-to-face meetings, gestures, eye contact), is tightly interwoven with and affected by the context in which the collaboration occurs.

Integrating the computer-based components of collaborative work with the context in which the work occurs is essential for useful forms of policy: our computers do not exist in a vacuum.

The third sample policy ("I need to share my workspace with others during demo days") obviously depends on detecting some "real world" condition of the participants in the scenario.

The fourth sample policy ("If anyone calls for me, tell them I'm not here unless they've called with the new budget numbers") may seem gratuitous, but this is precisely the sort of thing that occurs regularly in the workplace. A user will delegate responsibility for access control to another human being. This delegate (a good assistant typically) must digest the salient situational variables and decide in real time whether or not to grant access. Today, users must delegate

tasks such as this to other people since machine-based access control systems do not have the intelligence or situational awareness to act properly in such situations.

In all of these cases, we see that we need information about people (workgroups, advisors, people with budgets), about tasks (thesis work), and the context or setting of the action (is it demo day). and that this information must be dynamically-updated to reflect changes in the state of the world. In more general terms, the presence of situational information about the state of the "real world" is essential in deciding whether or not to grant access to a particular piece of information. In fact, bringing situational awareness into the environment goes a long way toward changing a simple, low-level access control mechanism into a more flexible and generally-applicable policy system.

The characteristics of these scenarios indicate that a traditional access control system augmented with information about situational variables will provide a powerful infrastructure for describing coordination policy. More succinctly, policy can be defined as the dynamic application of access control rights in response to "real world" situations. This idea is the approach taken by Intermezzo: we combine the formality and speed of a classical access control system, specifically, access control lists on data objects, with a scheme for sampling the "instantaneous" state of the users' world. A higher-level awareness system drives the low-level foundation access control system. This approach supports forms of policy that are both light-weight, and responsive to the changing environment in which the collaboration occurs.

The next section describes the Intermezzo object model, and how awareness works in the system.

THE INTERMEZZO FRAMEWORK

Our policy system is built atop the Intermezzo collaborative infrastructure. Intermezzo provides coordination support to applications in the form of session management, awareness, and policy control. The system provides a foundation object model on top of which higher-level collaborative functionality is based. While this paper does not describe all the features of the Intermezzo environment, a high-level view is important for understanding how policy is implemented in Intermezzo. This section describes some of the relevant features of the system.

Data Storage and Replication

The Intermezzo framework supports a replicated, consistent, and serialized object storage facility. The fundamental data objects manipulated by Intermezzo are called *resources*. Resources are essentially collections of key-value pairs called *attributes* (each attribute has a name, called its key, associated with it, and stores some arbitrary piece of data). One important data type that attributes can store is *links*. Links are pointers to other resources, and are used to form resources that contain aggregations of other resources.

Authentication and Authorization

All resources in Intermezzo support the notion of ownership—thus, some user owns a particular piece of data. The system provides access control lists (ACLs) to restrict access to resources and attributes within resources. An ACL is a mapping that indicates, for any given user, what operations are allowed on a specific object—either a resource or an attribute. Allowable operations include reading an object, writing an object, removing an object, and testing for the existence of an object. A special NONE right disallows access. The identity of users of the system is securely established through digital signatures; identity is represented by a unique string called a *certificate*.

Application Development

A toolkit, or code library, is used by Intermezzo client applications to enable the rapid construction of applications, and to provide certain conventions for the use of data in the form of resources. A multithreaded server process coordinates clients at runtime. This server also supports remote execution of code downloaded from clients.

Awareness Services

One important service provided by Intermezzo is an *activity-based awareness monitor*. While applications can use resources as a shared data store for any purposes they require, most Intermezzo applications will publish a predefined set of resources to represent the activity of the user running them. This data model for awareness is enforced by the toolkit. An activity is explicitly represented as a *Activity* resource that contains links to three component resources.

The first component is called the *Subject* resource, and contains information identifying the user. The second component is called the *Verb* resource, and represents the task or application being run. The third component is the *Object* resource, and represents the focus of the particular task—the file or other artifact on which the task is operating. See Figure 1, below.

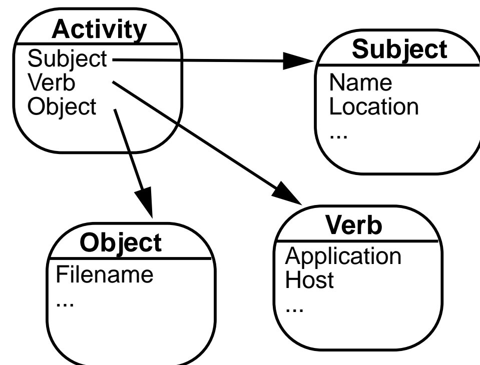

FIGURE 1: An Activity Resource.

Taken together, the set of resources published by Intermezzo applications provide a view of the distributed collection of users and their activities that is accessible to applications.

Several fundamental characteristics of this object model are used to create the policy system provided by Intermezzo: the basic object storage system, access control lists and authentication, and conventions for awareness.

ROLES IN INTERMEZZO

Policies are typically applied to sets or classifications of users. As noted earlier, the literature has adopted the term *role* to indicate a particular category of users with a set of access control rights applied to them, and in fact many collaborative systems have found the notion of roles useful for specifying how policies are assigned to users [8].

Intermezzo adopts the terminology of roles and applies the concept in a new direction. This section describes how policies are applied to roles.

Traditional Roles

Most systems that use roles have a few characteristics in common:

- The set of roles in use is determined *a priori* by the application or environment.
- The membership of those roles is typically determined early in the lifetime of the session.
- Membership of a role is specified in terms of users.
- There are few (if any) changes to role membership during the course of the session.

In traditional use, roles are typically *static* in the sense that membership of a given role is established early and rarely changed. Further (and perhaps obviously), a role is defined in terms of the users who are members of that role.

Such static roles are useful for a broad range of applications; but, as mentioned before, they lack flexibility and responsiveness to the environment in which they are used. Membership is predefined and fixed for certain access rights and the users themselves are responsible for updating role membership whenever policy mappings need to be changed.

In essence, users must digest the situational variables that are salient to coordination and use their world knowledge to specify a set of access rights. Such a system places a burden on the users to "model" the state of the collaboration. There are a number of factors that limit the utility of static roles in coordination:

- Static role systems require explicit overhead on the part of the user to set up the role membership. In fact, they require *anticipation* of potential coordination situations by users.
- Static roles ignore situational dynamics ("the real world") in deciding group membership.
- Specification is rigid: there is no flexibility or "slack" in the system.
- Role membership can only be defined in terms of user names, not other attributes of users or the environment.

The problems related to predefinition of role membership have been identified in the literature. Neuwirth, *et al*., have commented on the use of roles for collaborative writing [15],

> *There is a potential problem in systems which support the definition of social roles: "premature" definitions of these roles could lead to undesirable consequences. For example, it is not always clear at the outset of a project who is going to make a "significant contribution" and therefore who should get [the] authorship [role]. But if authorship is defined at the outset, then it may reduce the motivation of someone who has been defined as a "non-author" and the person may not contribute as much.*

Here we see a problem arising from static role definition in the application domain. And yet, in the PREP editor, early-defined roles are used, presumably because of the lack of a role infrastructure to build upon.

We have seen that static roles are useful: they provide a conceptual model for associating policies to users that is sufficient in many circumstances. They are well-understood by developers and users, and are efficiently implemented atop common access control mechanisms. Intermezzo provides static roles for applications that can benefit from them. But static roles are limiting in many collaborative situations because, as illustrated by the quote above, they cannot provide the flexibility needed in many situations.

Dynamic Roles

Intermezzo also supports a more flexible (although more expensive and somewhat harder to manage) form of roles that do not have the limitations of traditional static roles. These roles are called *dynamic roles*. The defining characteristic of dynamic roles is that *membership in the role is determined at runtime, as requests for access are made*. This characteristic means that dynamic roles have an important power that static roles do not: instead of *defining* roles in terms of their members, roles can be *described* in terms of their attributes.

Membership in a particular role is not determined by a membership list: instead it is determined by a predicate function that is evaluated whenever an access request is made. The use of potentially arbitrary predicates to determine membership lends great expressive power to dynamic roles.

By simply moving the determination of membership from session startup time to evaluation time, and by the use of predicate functions rather than membership lists, dynamic roles acquire several interesting properties:

- Dynamic roles allow role membership to be based on attributes other than "user name."
- Potential membership can vary from moment to moment during the lifetime of a session.
- Access can be granted based on the instantaneous state of the user's world.
- By describing role membership, rather than specifying it, users can be relieved of some of the burden of tracking, updating, and anticipating role membership explicitly.

As an example, via dynamic roles, you can not only specify "people who share my lab," but "*people who are in my lab right now*" as a role or category of users.

Dynamic roles extend one of the common threads in this research: that by bringing information about user awareness

into the collaborative environment, applications and the environment itself can be made more responsive to subtle changes in the state of the world. Further, users can be relieved of some of the burden of "manually" understanding and responding to situational variables; the system can take over some of that work when embodied with the required facilities and intelligence.

It is the dynamism inherent in the system—both in the dynamic application of users to roles, and in the constantly refreshed "world view" implemented by Intermezzo—that transforms a potentially rigid access control scheme into a more general policy system.

IMPLEMENTATION

This section describes the policy services provided by Intermezzo in detail. Intermezzo provides mechanisms for creating both static and dynamic roles, an implementation of policies on top of the "raw" access control system, and a specification language for roles and policies that can be used to control and configure the policy subsystem.

A Specification Language for Roles and Policies

Intermezzo provides a software substrate that implements static roles, dynamic roles, and policies in terms of access control. This "view" of this substrate from the perspective of applications and users is a declarative language that is used to specify policies and roles. The intent is that this language will be used by application developers and system administrators to configure policies and roles for their users.

At startup-time, the Intermezzo client-side code library loads a set of description files that contain specifications of policies and roles, and establish role-to-policy mappings. These specifications govern the access rights that are granted to resources created by that client. The predicate functions used to describe membership in a dynamic role are expressed in an extended version of the Python language [22].

Specifying Roles. The Intermezzo roles specification language supports three types of role specifications. Each role has a symbolic name associated with it that is used to bind the role to policies.

- Simple
- Dynamic
- Aggregate

Simple roles map a certificate name to a symbolic role name. These simple specifications are used to denote static roles. Recall that certificates are strings used throughout Intermezzo to represent users:

```
role keith = "Keith Edwards
             <kedwards@parc.xerox.com>"
```

Dynamic role declarations bind a predicate function to a symbolic role name. Predicates are specified by providing a path name on the client to the location of the code file that implements the predicate.

```
role demoday = [~keith/.policy/demoday.py]
```

Aggregate roles provide a mechanism for grouping other roles, whether simple, dynamic, or aggregates themselves. Aggregate roles can be nested arbitrarily deeply in a directed acyclic graph:

```
role myProj= {keith, beth, doug, demoday}
```

There is one role namespace within each parse-unit of the specification language. Note that inheritance of roles may be a useful feature. Inheritance would support the definition of classes of users, with specialization through derivations. This feature has not been explored in Intermezzo, however.

Specifying Policies. Policies are specified declaratively and, like roles, are named. A policy specification consists of a set of access control specifiers that denote the access rights that will exist for the set of resources and attributes referred to by that policy.

For purposes of assigning access rights, attributes are named by their key (the name of the attribute as represented by a string). Resources are named by type. Each data object (either resource or attribute) can have default or fallback access rights associated with it, which will be used if no more specific access right is provided.

Below is an example of a simple policy specification:

```
policy Restricted {
     resource Subject {
         attr Name = READ
         attr Location = WRITE
         attr * = NONE
     } = EXIST
     resource Verb = NONE
     resource * = EXIST
}
```

This specification creates a new policy named *Restricted*. The policy provides two by-type resource specifications for *Subject* and *Verb* resources. For any user that belongs to a role using the Restricted policy, Subjects will have the EXIST access right (allowing the existence of the resource to be tested), and Verbs will have the NONE right (denying all access). The wildcard denotes that the policy associates the EXIST right with all resources of types not explicitly named.

The policy also provides two access specifiers for attributes with keys *Name* and *Location*. Name attributes have READ rights (allowing viewing of the data); Location attributes have WRITE permissions (allowing updates of the data). Note that since the attribute specifiers are nested within the specification for resource Subject, these rights will only be used for Name and Location attributes that are present in Subject resources. The wild card specifier indicates that any attributes in Subject resources that are not explicitly named by the policy inherit the NONE right.

Note that the example here shows a policy that applies access control to the set of resources used by Intermezzo for awareness. Hence, this policy restricts access to the information used to coordinate among groups of users. In general, however, an access control policy can be applied to *any* data object stored by Intermezzo. Hence, if a

collaborative editing system uses resources to store shared data, access to that data could also be regulated by a policy.

Mapping from Roles to Policies. The language for specification also provides a means for establishing mappings between roles and policies. Symbolic role names are mapped to symbolic policy names; the system supports many-to-many mappings.

```
gvu_lab          -> RestrictedAccess
animation_lab    -> RestrictedAccess
friends          -> GeneralAccess
stasko           -> AdvisorAccess
*                -> Anonymity
```

Note the use of wildcard specifications for roles: the language allows a default policy that will be applied to all users not represented by any of the provided roles.

Many-to-many mappings provide two benefits. First, they allow several roles to share the same policy, resulting in reuse of policy descriptions. In the example above, the policy *RestrictedAccess* is reused by two different roles: *gvu_lab* and *animation_lab*.

The second benefit of many-to-many mappings is that they allow multiple policies to be associated with a given role, allowing "segmenting" of access rights across multiple policies. In this case each policy would specify a set of access rights for a different set of objects. Thus, ideally there would be no "overlap" between the access rights specified by multiple policies associated with a particular role.

A static analysis could detect policies that do overlap, although this facility is not implemented currently. In practice, however, the presence of dynamic roles means that the system cannot know until runtime whether an actual overlap of access rights for a given user exists. Detecting and disallowing such overlaps late (at runtime) seemed overly restrictive and cumbersome.

If a user is in multiple roles simultaneously, Intermezzo uses a liberal policy for assigning effective access rights: the strongest right allowed by any policy of which the user is a member is applied. This rule supports "layering" of policies on a given user, each granting wider access.

Static Role Implementation

The notion of static roles is constructed by aggregating features found in the Intermezzo foundation layer: static roles are implemented using simple access control lists. In fact, the notion of a "role," whether static or dynamic, is present only in the higher-level coordination features provided by Intermezzo (the parser for the specification language, for example).

When an Intermezzo client is run, the role specifications are parsed at application startup time. The client-side toolkit "digests" these specifications into a compiled format that specifies the access control lists that resources and attributes will inherit. This processed format is essentially an inverse of the role (user) to policy (access control) specification supported by the specification language. The system builds a list of all of the resources and attributes that are specified in the policy descriptions. It then "works backwards" to find all roles that map to policies that mention these resources and attributes. From this information it is possible to construct a set of "prototype" access control lists. These prototypes are mappings from resource types or attribute names to the access control lists that will be associated with those resources or attributes when they are created.

This mapping is maintained internally by the client-side library. Whenever the application creates a resource, or an attribute on an existing resource, the access prototype mapping is queried to retrieve an access control list to be applied to the new data object. Via this scheme, the penalty of parsing and internalizing the policy descriptions is paid only once, at application startup time. After startup, the internalized policy descriptors are used to automatically generate the access control lists for resources and attributed created by the application. The "generation" of access control lists is essentially just a table lookup in the prototypes mapping, and is thus quite inexpensive.

Since static roles are only manifested as access control lists at runtime, determination of access rights is trivial. There is no need to search role membership lists to determine access; instead, the system merely retrieves the access right associated with a given user when that user attempts to access an object. Determination of access at runtime is as simple as a dictionary lookup to retrieve the right from the access control list, and then an arithmetic comparison to determine whether the right will be granted.

Note that the owner of a resource always has full rights to it. Updates and reads of resources by their owners is the common case and bypasses the access control mechanisms. Since ownership is authenticated this poses no security risk.

Dynamic Role Implementation

Whereas static roles are implemented by associating a list of users with a set of access control rights, dynamic roles are implemented by associating a predicate function with a set of access control rights. When a client starts, it generates a "map" of access rights for the resources and attributes that are represented by the policies it loads. In dynamic roles, each resource and attribute keeps a list of access rights and predicate functions that map onto those rights.

When a request for access is made at runtime, Intermezzo first evaluates the access control list for the object to see if the requested right is explicitly granted to the user making the access. If it is not, then the system scans the list of access rights associated with predicate functions. For any access rights that provide the requested access, the system evaluates the associated predicate functions until either a predicate returns true or no predicates remain that might be able to grant the requested access.

In essence, this scheme means that the system first applies the static roles to see if access may be granted. Only if they fail does the system apply dynamic roles. Further, only the predicate functions that have the potential to grant access are evaluated, and they are only evaluated until the access is granted. This is consistent with the "liberal" application of the access control system throughout Intermezzo, and provides efficient short-circuit evaluation: if *any* policy

would grant access to a particular object, then access is granted.

Predicates are evaluated in the server and execute with the permissions of the user requesting access. Evaluation occurs in the server because the alternative, execution within the client, would be a security risk: rogue clients could claim to have executed the predicate and return a successful condition to the server. Predicate execution occurs with the user's permissions to ensure that a predicate function cannot be used as a "Trojan horse" to gain access to resource data that would otherwise not be accessible to a given user.

Below is an example of a predicate function expressed in the form of extended Python used by Intermezzo. This predicate defines a role for the "demo day" scenario discussed earlier in this paper. There are a number of possible ways to define a "demo day" predicate; this one considers the following characteristics salient:

- The day must be demo day.
- Only extend access when the owner ("me") is in the lab.
- The user must be running the demo application ("Montage").
- Access is only extended to those who are themselves in the lab.

```
def predicate(subj me context):
    if date.today() == "August 28":
        if me.Location = "GVULab":
            if me.Activities("Montage"):
                if subj.Location == "GVULab":
                    return TRUE

    return FALSE
```

The predicate consists entirely of a set of conditions to determine (1) the date, (2) the location of the resource owner, (3) the current activities of the owner, and (4) the location of the requesting user.

Note the parameters passed to this function: `subj` is a resource of type Subject that represents the user requesting access. The predicate is free to query the attributes of this resource, modulo any access constraints placed on it, to determine information about the requesting user. The `me` argument is also a Subject resource that represents the owner of the data object to which access is being requested (the predicate itself is executed with the permissions of this user). The final `context` argument represents the world-view collection of resources that represents the state of the awareness service. Via this parameter, the predicate is free to query about any condition in the Intermezzo data model.

While some effort is made to speed dynamic predicate evaluation (by evaluating static roles first, and by shortcut evaluation, for example), there are other optimizations that are possible but are not implemented currently. First, predicates can be cached on the server; clients would then only transmit an identifying token for the predicate, rather than the predicate itself.

Second (and more difficult), we may be able to pre-process a predicate to determine which objects are examined by it. A predicate result can only vary between executions if the "world state" on which it depends has changed.

COMMON POLICIES

While the policy system used by Intermezzo is flexible and powerful enough to capture a range of specific and complex "real world" situations, it is also useful for describing a number of common general policies. This section describes two general policies which seem to be not only highly useful, but also "in demand" by users. Again, note that these are coordination policies, as they regulate access to the predefined resources used to enable awareness and coordination among users.

Anonymity

In some collaborative situations it may be useful to prevent broad access to a set of information about users. For example, in brainstorming sessions such as those provided by group decision support systems, research has shown that often better results are achieved if the participants do not know who is submitting ideas [16]. By keeping participants from being able to identify one another, social barriers to contributing to the session are lowered.

This sort of information restriction is called *anonymity*. There is no information associated with users that could be used to identify them. Further, it is impossible (or at least unlikely) to track user data over a long period of time in an attempt to glean enough information to identify a user.

Intermezzo makes the construction of policies that correspond to anonymity possible. The precise definition of such policies will vary from site to site because of the need to restrict different pieces of information based on common user behaviors and tasks. For example, at a site where users are typically mobile, it may be desirable to allow access to information about location ("I know someone's in the coffee room, but I don't know who it is.") At a site where users typically sit at their desks all day, information about location is in practice equivalent to direct information about user identity.

Below is one (perhaps overly simple) definition of a policy to support anonymity.

```
policy Anonymity {
    resource Subject = {
        attr * = NONE
    } = EXIST
    resource Verb = {
        attr * = NONE
    } = EXIST
    resource * = NONE
}
```

This policy does not allow a user to read or even determine the existence of any attributes on the *Subject* or *Verb* resources. It does allow the determination of the existence of these resources as a whole, however, and it does allow access to resources of types other than Subject and Verb.

The anonymity policy as defined above is not the same as "invisibility:" invisibility would remove the EXIST rights from resources, preventing detection of even the presence of

users. Anonymity is weaker in that it permits questions like, "How many users are logged on now?" and, "What applications are being run?" while still removing identifying characteristics from that information. Still, however, anonymity is so restrictive that it prevents many of the activity-based interaction features of Intermezzo from operating. For example, the Intermezzo session management service uses attributes on the Subject and Verb resources. If these attributes cannot be read, the session management system will be unable to rendezvous with a user. (For a more thorough description of how session management interacts with the activity service, see [6]).

Pseudonymity

Often, anonymity is too restrictive in collaborative situations. While it is useful in certain constrained environments (such as the brainstorming example), it severely limits the flow of information that may be useful to coordination. *Pseudonymity* is a policy that protects privacy but still allows access to information that can support coordination between groups of users.

Pseudonymity is similar to anonymity in that it does not allow information published in the shared information space to be associated with an actual human user of the system. It *does* however allow the use or identifying "handles" that can be used to track users in the abstract. Pseudonymity allows questions like, "What is user X's typical work flow?" and, "Which users that run Framemaker also run Photoshop?" Pseudonymity supports the collection of user statistics and allows the system to track the paths of individual activities, while preventing those activities from being associated with an actual person. A common analog in the social sciences is the use of code numbers or names for experimental subjects. These codes are typically used over the course of a long-lived experiment (sometimes over a period of years), but still provide no information about the true name of "User X."

Again, like the implementation of anonymity, a policy for pseudonymity may vary from site to site. At some sites it may be possible to release certain pieces of information while preserving pseudonymity; at others, more information may be restricted.

Below is an example of a policy for pseudonymity:

```
policy Pseudonymity {
     resource Subject {
         attr Location = READ
         #...other attributes we may wish to
         # allow access to...
         attr * = NONE
     } = EXIST
     resource Activity {
         attr * = READ
     } = EXIST
}
```

This policy varies from anonymity in a number of respects. First, it selectively allows access to certain attributes of the *Subject* resource (*Location* in this example). Next, it allows read access to the attributes of *Activity* resources. Allowing reads of Activities allows applications to *read the links* (meaning, the actual hashed resource identifier) in the Activity slots for Subject and so on, yet prevents reading the actual resources pointed to by these links (since Subject only supports limited read access).

By allowing applications to read the links, but not the data, in an Activity resource, applications can detect when several Activities "point" to the same user, while denying access to potentially identifying information about that user.

Pseudonymity is useful for the (perhaps very common) case of access control where a user may say, "I don't mind people knowing what I'm doing, as long as they don't know it's *me* doing it." Note that users with pseudonymity enabled are able to interoperate with the Intermezzo session management services since the policy does allow access to activity and some user information.

SUMMARY AND FUTURE DIRECTIONS

The ability to describe to a system how it should behave in a potentially chaotic setting is important for collaborative work. In this paper, I have used the term policy to denote a set of access control rights dynamically applied to data objects, and mediated by the situational context in which the collaboration occurs. The data objects can represent user activity and awareness, as well as application-specific data. This access control-based definition is capable of capturing many useful policies, from the specific (how to deal with particular users at particular times of the day) to the general (anonymity and pseudonymity).

The system uses the notion of roles to associate categories of users with particular policies. Intermezzo roles can represent not only groups of users, but also descriptions of users in the form of predicates evaluated at runtime to determine group membership.

Dynamic roles, in particular, expand on one of the central themes in this work: by bringing information about users and their environments into the system, we can make computer-augmented collaboration more responsive, and we can free the users of many of the burdens implicit in working with today's collaborative systems.

There are a number of areas for future work. One limitation of this work is the policy specification language. The language is powerful, and provides an economical way to create new policies and roles for applications—previously developers had to "hard code" support for policies and roles. The problem is that while the language is much easier than developing code, it is still not suitable for end-users. Ideally users should be able to create their own policies and roles as the need arises. Users should be able to selectively enable or disable access control without having to learn a new language, running the risk of inadvertently exposing their data to unwanted access, or understanding the intricacies of the resource object model and access control rights.

A "policy manager" tool that interacts with users and emits policy specifications according to user desires would be an interesting avenue of research. In fact, the entire issue of how end-users *think* about policies and roles, and how to capture those concepts in a tool designed for end users, would make an interesting area of study.

REFERENCES

[1] *The American Heritage* Dictionary. Boston, MA: Houghton Mifflin Company.

[2] Baecker, R.M., Nastos, D., Posner, I.R., and Mawby, K.L., "The User-centered Iterative Design of Collaborative Writing Software." In *Proceedings of the ACM/InterAct Conference on Human Factors in Computing Systems*. Amsterdam, The Netherlands: ACM. April 24-29, 1993. pp. 399-405.

[3] Brothers, L., Sembugamoorthy, V., Muller, M., "ICICLE: Groupware for Code Inspection." *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, pp. 169-181.

[4] Dewan, Prasun, Choudhary, Rajiv, and Shen, HongHai, "An Editing-based Characterization of the Design Space of Collaborative Applications." *Journal of Organizational Computing*, 4:3, pp. 219-240, 1994.

[5] Dourish, Paul, and Bellotti, Victoria. "Awareness and Coordination in Shared Work Spaces." *Proceedings of ACM Conference on Computer-Supported Cooperative Work*, Toronto, Canada, November 1992.

[6] Edwards, W. Keith, "Session Management for Collaborative Applications." In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, Chapel Hill, NC: ACM, October 22-26, 1994. pp. 323-330.

[7] Edwards, W. Keith, *Coordination Infrastructure in Collaborative Systems*. Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, GA. November 22, 1995.

[8] Gintell, John W., and McKenney, Roland F., "CSCW Infrastructure Requirements Derived from Scrutiny Project." Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work, Chapel Hill, NC October 22, 1994

[9] Grief, I., and Sarin, S. "Data Sharing in Group Work," *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 477-508.

[10] Kaplan, S.M., Tolone, W.J., Bogia, D.P., and Bignoli, C., "Flexible, Active Support for Collaborative Work with ConversationBuilder." *Proceedings of the Conference on Computer-Supported Cooperative Work*, Toronto, Ontario: ACM, pp. 378-385.

[11] Leland, M.D.P., Fish, R.S., and Kraut, R.E., "Collaborative Document Production Using Quilt." *Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, 1988, 206-215.

[12] Mantei, M.M, Baecker, R.M., Sellen, A.J., Buxton, W.A.S., Milligan, T., and Wellman, B. "Experiences in the Use of a Media Space." *Proceedings of the ACM Conference on Computer-Human Interaction*, April 28-May2, 1991. New Orleans, LA: ACM. pp. 127-138.

[13] Moffett, Jonathan D., and Morris, S. Sloman, "The Representation of Policies as System Objects." In *Proceedings of the ACM Conference on Organizational Computing Systems*, Atlanta, GA: ACM, November 5-8, 1991, pp. 171-184.

[14] Moran, Thomas P., and Anderson, R.J., "The Workaday World as a Paradigm for CSCW Design." *Proceedings of the Conference on Computer-Supported Cooperative Work,* Los Angeles, CA: ACM, 1990, pp. 381-393.

[15] Neuwirth, C. M., Kaufer, D. S., Chandhok, R., and Morris, J. "Issues in the Design of Computer Support for Co-authoring and Commenting." *Proceedings of the Conference on Computer-Supported Cooperative Work,* Los Angeles, CA: ACM, 1990, 183-195.

[16] Poole, M.S., Holmes, M., and DeSanctis, G., "Conflict Management and Group Decision Support Systems." In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM. September 26-28, 1988, pp. 227-241.

[17] Root, R.W. "Design of a Multi-Media Vehicle for Social Browsing," *Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, September 26-28, 1988. pp. 25-38.

[18] Shen, H., and Dewan, P. "Access Control for Collaborative Environments," *Proceedings of the Conference on Computer-Supported Cooperative Work*, Toronto, Ontario: ACM, 1992, 51-58.

[19] Sohlenkamp, Markus, and Chwelos, Greg, "Integrating Communication, Cooperation, and Awareness: The DIVA Virtual Office Environment." *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, Chapel Hill, October 22-26, 1994. pp. 331-343.

[20] Smith, Ian, and Hudson, Scott, "Low Disturbance Audio For Awareness And Privacy In Media Space Applications," *Proceedings of ACM Conference On Multimedia*, November, 1995, San Francisco, CA: ACM.

[21] Tang, J.C., Isaacs, E.A., and Rua, M., "Supporting Distributed Groups with a Montage of Lightweight Interactions." *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, Chapel Hill, October 22-26, 1994. pp. 23-34.

[22] Van Rossum, Guido, *Python Reference Manual Release 1.3*. October 13, 1995 (available as http://www.python.org/doc/ref/ref.html).