

# 1

---

## *A Coordination Model for Secure Collaboration*

Anand Tripathi  
Tanvir Ahmed  
Richa Kumar  
Shremattie Jaman

The focus of this paper is on specifying a coordination model for building secure distributed collaboration and workflow systems from their high level specifications. We identify here unique requirements of security and coordination in dynamic collaboration environments. We present a role-based model for specifying these requirements. This specification model supports hierarchical structuring of a large collaboration environment using nested activities, which can be created dynamically. We briefly describe how a middleware is used to realize and support a collaboration environment from its specifications, enforcing the required policies for coordination and security.

### **1.1 INTRODUCTION**

In computer supported cooperative work (CSCW) systems, multiple users cooperate and collaborate towards some shared objectives and tasks using shared data and communication channels [6]. The fundamental challenges of a CSCW system include sharing data and coordinating activities among the collaborating participants. Coordination in a CSCW environment can range from tightly coupled real-time interactions, where users are connected

to the system simultaneously and interact through sharing and manipulation of graphical and multimedia objects, to loosely coupled interactions, which are largely characterized by workflow environments where the coordination is coarse-grain.

These challenges warrant an increasing awareness of security issues. With collaboration activities possibly spanning different organizations and multiple users concurrently accessing shared objects, there is a need to maintain confidentiality, consistency, and integrity of the shared data. Often, security and coordination are co-dependent in a collaboration environment.

Our goal is to provide a specification language that can express the coordination and security requirements for a range of collaboration systems. From this specification, we can rapidly construct the runtime environment for the system by means of a generic policy-driven middleware [23]. In our model, a policy-driven collaboration system is realized in three steps. Initially, the coordination and security policy for a collaboration is specified based on a schema. From the specification, various policy modules are derived for different kinds of requirements, such as role based security, object level access control, and event notification for coordination. Finally, through these modules, the collaboration environment is realized by a generic middleware.

A policy-driven system is able to specify the dynamic nature of CSCW activities and enforce coordination and security policies at runtime. Existing policy driven CSCW systems, such as COCA [12] and DCWPL [2], are built mainly for shared interactive groupware applications. These systems are for time-space limited activities, and therefore the models for describing the desired systems cannot implement large workflow like CSCW environments where multiple CSCW applications may come into existence within a single collaboration framework.

Our model centers around role-based policies for security and coordination [16, 13]. This model is often used in collaboration systems where participants perform a set of well-defined tasks pertaining to their qualifications and responsibilities in the organization. All privileges are granted to a role rather than a user. A role-based model allows a role to exist independent of its members and provides a means for associating privileges for task execution with the various participants in the system.

A distributed collaboration has a number of coordination and security requirements. These include

1. Hierarchical activity organization
2. Dynamic security model
3. Role management
4. Coordination model

**Hierarchical activity organization:** A large collaborative environment may sometimes need to be structured hierarchically, consisting of smaller activities nested inside it. For example, a system supporting a team based design project running over several weeks or months may need to have several dy-

namically created “whiteboard-like” activities of relatively shorter life-spans. Therefore, a facility is needed to hierarchically structure a large CSCW environment into nested sub-activities. Our specification model supports hierarchical and dynamic structuring of a collaboration system using the notion of an *activity*, which defines a protection domain and scoping facility, encapsulating a set of objects, roles, and policies for security and coordination. An activity can have nested sub-activities and multiple instances of the same activity can exist concurrently.

**Dynamic security model:** The traditional concerns of confidentiality and integrity of shared data are naturally present in collaboration systems. However, access rights, privileges, and ownership of objects may change in collaborative environments as activities progress. Access control may need to be history-based, where the privileges depend on past events. The privileges assigned to a user in a role may change with time due to the actions executed by other participants. Sometimes permissions may change due to the user’s own actions. Dynamic access control policies also have to address the issues of role invalidation, role activation, or other conditions that may occur in the system. Privacy can also become an issue when one may need to hide the identity of one participant from another. In such cases, the presence of a participant may only be visible through his/her role or a pseudonym in a role but not by name. The principle of “separation of duties” is an important requirement in a business or office environment [17, 18, 21]. One individual should not be allowed to perform all critical functions in a business transaction. Instead, such responsibilities should be shared among different users to eliminate conflict-of-interest.

**Role management:** A role represents a set of privileges and can be viewed as a characterization of a protection domain. In role-based systems, static assignment of participants to roles can be error prone and such assignments do not scale well. In a large collaboration environment there is a need for dynamic assignment and delegation of roles. *Role admission constraints* and *activation constraints* address this issue and are described in Section 1.3. Participants may join/leave a role or alternatively be admitted/removed from roles. During the lifetime of a role, a list of the current members of the role needs to be maintained. As participants join or are admitted to a role, various checks need to be performed to conform to the security policies. Similar checks need to be done when any task is performed by a participant in a role.

**Coordination model:** Coordination among participants in different roles within an activity is based on events generated by the actions of the collaborating participants. For some roles, multiple users may be allowed to simultaneously acquire and activate the role. Coordination among such users can be referred to as *intra-role* coordination. Participation of multiple users in a role can be *independent* or *cooperative*. In independent participation, every member assumes responsibility for certain role-specific tasks, irrespective of the presence of other participants. For example, in a document authoring activity having three members in a *Reviewer* role, each member writes an

independent review. When participants coordinate and share the tasks of the role, it is referred to as *cooperative* participation. For example, in a medical system, certain procedures might be required to be performed on a patient by any one nurse. Another form of intra-role coordination requires that all users in the role must participate for a certain operation to be successful. An example would be a group of participants jointly opening a bank vault. Some scenarios may also warrant less rigid coordination models, e.g. in an unrestricted whiteboard sharing activity.

This paper presents a specification model which addresses the above security and coordination requirements within hierarchically nested collaborative systems and which facilitates the derivation of policy modules by a generic middleware to support the runtime environment.

Based on this model, we have devised an XML schema, which facilitates collaboration specifications. We have developed a middleware that implements the XML based policy specifications to realize a collaboration environment. This approach allows one to easily install and experiment with new policies. It also allows us to perform some static checks for consistency in the specifications.

An overview of our collaboration model is presented in Section 1.2 with an example specification of a collaboration environment. Section 1.3 describes how roles, objects and activities are specified in our model. An overview of the middleware execution model is discussed in Section 1.4. Sections 1.5 and 1.6 discuss the related work and the conclusions.

## 1.2 OVERVIEW OF THE COLLABORATION MODEL

In our model, an *activity template* specifies a generic pattern for collaboration among a set of roles using some shared objects. Any number of instances of it can be dynamically and concurrently instantiated. There are three major elements in specifying a collaboration activity: shared objects, roles, and operations. These entities are specified and named in nested scopes of activities as shown in Figure 1.1. Every activity instance has two meta roles: *Creator* and *Owner*. The user instantiating the activity becomes a member of the *Creator* role. However the owner of the activity may be different from its creator and can be specified in the activity's definition. If not specified, the owner of its parent activity becomes the default owner. The owner of an activity also owns the encapsulated roles, and is trusted for the management of that activity and its nested entities.

### 1.2.1 Shared Objects

Shared objects are managed by a set of trusted object servers, which are responsible for protecting the objects according to security policies. Each activ-

ity can designate an object server to be used for storing all the objects created within the scope of the activity. Based on the security and coordination requirements specified in a collaboration, our system derives appropriate policy modules, similar to the adapter approach [22], which are used by the servers to control access to their objects. The server is also responsible for managing recovery and persistent storage of its objects. Caching and replication however are implementation issues and are transparent at the specification level.

The object servers in our system support both role based access control as well as traditional discretionary access control (DAC). Access control lists are maintained based on user-ids as well as role names. Access control policies can be specified for each method of an object or the object itself. Negative access rights (i.e., constraints) enable easier coarse-grain access control specification.

### 1.2.2 Role Management

In defining a role, we consider the constraints addressed in Section 1.1. These constraints are of three types: the *role admission condition* puts constraints on admitting a participant to a role, the *role activation condition* needs to be satisfied when a role is activated, and the *precondition* of an operation is required to be satisfied for a member in the role to execute it. The owner of a role can admit users to it, subject to the admission constraints; it can also remove an existing participant. The constraint specifications can include event counters. We present details of these constraint specifications in the next section and show how the security requirements discussed in the previous section can be realized.

The specification for a role in an activity can refer to the objects and other roles in that activity. A nested activity may need to have access to the objects in the scope of its parent activity, or a role in the parent activity may need to be assigned to a role in a nested activity. For this purpose, an activity definition needs mechanisms for role assignment and passing of object references as parameters to an activity instance. A role in the parent activity can be assigned to a role in the child activity in the activity's definition. We refer to it as *role reflection*, which means that all the members of the parent role implicitly become members of the role in the child activity. Removal of a participant from the reflected role (i.e. a role in the parent activity), also implies removal from the role in the child activity. A role reflection can also be specified at the time of activity instantiation. A participant in the reflected roles gains expanded privileges comprising of the operations of the child activity role. Moreover, the child activity role can have operations that access the objects in the scope of the reflected role. However, any participant of the reflected role has to comply with the child role's admission constraints.

An operation in our model represents a task of a role. An operation may consist of the execution of a method of a shared object, a synchronization action, or a role request. A role request is an operation either to *leave* the

current role or to *join* another role. Coordination policies are specified as preconditions for operations using an event based model [15].

In the specification model several functions are defined for a role. A boolean function *member(role, user)* checks if a participant is present in a role. The function *members(role)* gives the list of participants in a role. We provide a count operator *#* on lists. Hence, a count of the participants admitted in a role is given by  *#(members(role))*.

### 1.2.3 Events

Events signify state changes in a collaboration environment. Events and event counters are used in our model for specifying coordination and dynamic security policies. Event types are based on conditions related to different kinds of entities such as activities, roles, operations, and objects. For example, instantiation of an activity, execution of a role operation, admission of a user in a given role etc. represent different type of events. An event can be identified using the scope based naming. Each event is associated with some collaboration object or some meta object, like a role or activity instance. All events are derived from the base event class, which contains certain common attributes such as *type, id, creation-time, activity-id* etc.

### 1.2.4 An Example of a Collaboration Activity

The example collaboration environment considered in this paper supports the activities of a group of people jointly developing a document. This will serve as a detailed example to illustrate how policy modules are derived from the specification. Through this example, we highlight some of the coordination and security requirements that were described in Section 1.1 and provide mechanisms for expressing such requirements through our specification model. Figure 1.1 introduces the overall model of our authoring example specified as a *DocumentAuthoring* activity template. The figure shows an instance *document* of this template. The template encapsulates the central entities of the collaboration: a shared document, roles, and collaboration constraints. Authoring of the document object composed of multiple chapter objects is the objective of this collaboration. Three roles are defined for the authoring activity: *Author, ExternalReviewer, and Supervisor*. In the instance of this *DocumentAuthoring* activity template illustrated in the example, the *Supervisor* role can be referred to by its fully qualified name *DocumentAuthoring.document.Supervisor*.

In the *DocumentAuthoring* activity, the *ChapterAuthoring* activity is defined as a nested activity with three roles: *Author, Reviewer, and Editor*. The document can be composed of any number of chapter objects, and an instance of the chapter object is passed to a *ChapterAuthoring* activity. The example in Figure 1.1 shows two instances of this template: *chapter1* and *chapter2*.

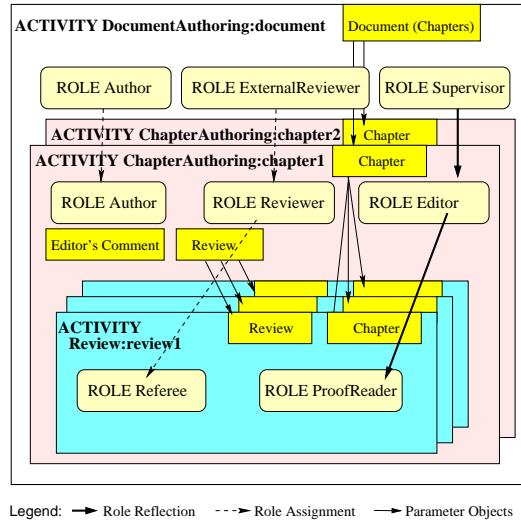


Fig. 1.1 Hierarchical Structuring of Collaborative Activities

A chapter authoring activity manipulates three objects that are shared: the chapter’s content, the reviewer’s review, and the editor’s comments. This *ChapterAuthoring* can be represented as a task-flow diagram, shown in Figure 1.2. Within the *ChapterAuthoring* activity, a participant in the *Author* role writes the contents of the chapter and publishes them. A participant in the *Reviewer* role can then create a *Review* activity in order to prepare an independent review. Two roles, *Referee* and *ProofReader*, are defined for the *Review* activity. When all the reviewers have made their reviews available to the editor, the editor publishes his/her comments, which can be read by the author. Based on the editor’s comments, the authors may modify the chapter’s content and publish the final version.

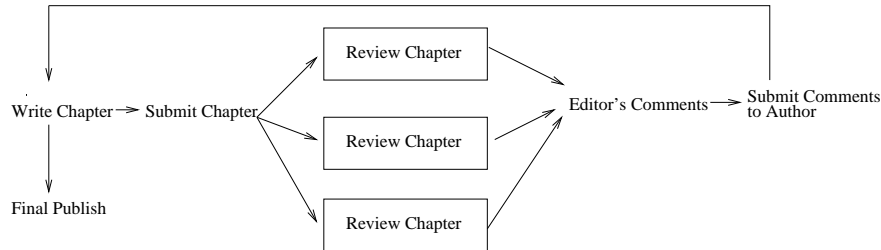


Fig. 1.2 Task flow diagram of ChapterAuthoring Activity

In our example, a participant cannot simultaneously be in the *Author* and the *ExternalReviewer* roles of the *DocumentAuthoring* activity. The *Supervi-*

role of the *DocumentAuthoring* activity is reflected in the *Editor* role of the *ChapterAuthoring* activity. When an instance of the *ChapterAuthoring* template is created, e.g. *chapter1*, the members of the *Supervisor* role who comply with the *Editor* role's admission constraints are admitted to this role. Therefore, an admitted participant in the *Editor* role can perform operations on the *Document* object in the *DocumentAuthoring* activity as well as on the objects in the *ChapterAuthoring* activity. Similarly, the participants in the *Editor* role are reflected in the *ProofReader* role of the nested *Review* activity. A participant in the *Author* role of the *ChapterAuthoring* activity must be a member of the *Author* role of the parent activity. Similarly, there are certain constraints on the membership of the *Reviewer* role. There may be at most three members in this role, with one member being an author in the parent *DocumentAuthoring* activity but not in the current instance of the *ChapterAuthoring* activity, and the other two being external reviewers. The editor can publish his/her comments only after the chapter's contents have been independently reviewed by three reviewers.

### 1.3 SPECIFICATION MODEL

#### 1.3.1 Event Specification

Related to each operation are two types of events: *start*, and *finish*. For example, *role-name.op.start* or *role-name.op.finish*. These event types are also defined for each object method. The corresponding event counters can be used for synchronization. For a role, we have event types defined for *join*, *leave*, *admit*, and *remove* operations.

Multiple occurrences of a given event type, such as multiple executions of an operation, are represented by a list. The expression  $(eventName)$  returns the list including all the instances of this type of event. Hence,  $\#(eventName)$  returns the number of times the event has occurred.

In our model, one can also specify a derived event type by filtering an event list based on its attributes. For example, for a role operation execution, we can define a filter based on invoker id, such as `opName.start(invoker=John)`. We can count how many times a user has invoked an operation using `\#(opName.start(invoker=John))`. An example of a role related event is `DocumentAuthoring.Author.join(user=Mary)`.

#### 1.3.2 Object Specification

Shared objects are represented in our specification model only in terms of their types and method signatures, keeping the semantics and implementation details transparent. The method signatures are used in specifying the operations associated with a role. A specification for an object includes a

type name to facilitate parameter bindings of operations in roles, a codebase to load the class of the object, and method signatures.

There are three distinct kinds of objects used in our specification: regular objects utilized in a collaborative workspace, roles, and activities. We can specify access control at the granularity of the methods invoked on these objects. User level access control is specified in the scope of a role as part of operation specification. The object level access control specification is derived from the role's privileges on objects. After an object is created the owner can specify additional object level access control. Given below is the specification for the *Chapter* object type in our example. Definition for *ContentType* is not shown.

```
OBJECT_TYPE Chapter (CODEBASE=http://codeserver) {
  METHOD writeContent {PARAM ContentType}
  METHOD readContent {RETURN ContentType} }
```

### 1.3.3 Role Specification

A role is defined in the scope of an activity and can manipulate the objects in that activity. A role specification includes role name, reflected roles if any, role admission constraints, role activation constraints, and role operations with their preconditions. The basic terms of a role specification are shown below, where [ ] encloses optional terms.

```
ROLE name ([REFLECT role]) {
  [ADMISSION_CONSTRAINTS condition]
  [ACTIVATION_CONSTRAINTS condition]
  [OPERATION name
   [PRECONDITION condition]
   [ACTION method_invocation]]}
```

**1.3.3.1 Role Admission Constraints** Role admission constraints are enforced when a participant is assigned or admitted to the role. These constraints enforce several “separation of duty” policies, specifically *static separation of duties*, *user-user conflict*, *user-role conflict*, and *history-based* conditions [18, 14].

In our example, in order to ensure that no participant is present in both the *Author* and the *ExternalReviewer* role of the *DocumentAuthoring* activity, the *ExternalReviewer* role has an admission constraint that a participant in this role cannot be a member of the *Author* role within the same activity. This is specified as follows:

```
ROLE ExternalReviewer {
  ADMISSION_CONSTRAINTS
    !member(thisUser, Author)
}
```

Similarly, the *Author* role has an admission constraint that specifies that the participant should not be a member of the *ExternalReviewer* role. These two together form an example of a static “separation of duties”.

Role admission constraints can take various forms, for example, a list of valid or invalid users, role operation based conditions, prerequisite roles, or cardinality of the role specifying maximum member count.

**1.3.3.2 Role Activation Constraints** Role activation constraints are enforced when a user invokes a role operation. In contrast, role admission constraints are checked only when a user is admitted to a role. Dynamic “separation of duty” constraints can be specified as part of role activation constraints. For example, the static “separation of duties” requirement for the *Author* and *ExternalReviewer* roles in the *DocumentAuthoring* activity, which is specified by means of admission constraints in Section 1.3.3.1, can be changed to be enforced dynamically using activation constraints as follows:

```
ROLE ExternalReviewer {
  ADMISSION_CONSTRAINTS
    !member(thisUser, Author)
  ACTIVATION_CONSTRAINTS
    !member(thisUser, Author)
    .....
}
```

Assuming that, the *Author* role does not have any admission constraints, a participant who is not currently a member of the *Author* role can first join the *ExternalReviewer* role, satisfying its admission constraints, and later join the *Author* role. However, the activation constraints of the *ExternalReviewer* role ensure that whenever a member of this role invokes a role operation, the invoker is not a member of the *Author* role. In this way “separation of duties” is enforced dynamically.

Activation constraints may specify a minimum and a maximum number of participants to be present in a role. Additionally, activation constraints may specify some coordination requirements, for example, the activation constraints for the *Reviewer* role in the *ChapterAuthoring* activity, as shown in Figure 1.4, ensure that the reviewer cannot perform any operations unless the chapter has been submitted by the author.

**1.3.3.3 Operation Specification** An operation is specified by a name, an optional precondition, and an action which invokes predefined methods of some objects in the shared workspace. A keyword *new* is reserved for specifying object and activity creation.

Following is an example of an operation specification for the *Author* role in the *DocumentAuthoring* activity.

```

ROLE Author {
  OPERATION startChapterAuthoring {
    ACTION {chapter=new OBJECT(Chapter)
           act=new ACTIVITY ChapterAuthoring (
               chapter, Author=thisUser) }}
}

```

This specification does not include a precondition, but simply specifies the action that must be taken when the operation is performed. The action includes the creation of a new *ChapterAuthoring* activity, passing the shared *Chapter* object as well as assigning the current user to the *Author* role within the nested activity. In this way nested activities can be dynamically instantiated by performing role operations.

When a precondition is specified for an operation, it must be true when the operation is invoked. This allows us to specify a number of coordination and security constraints like *operational separation of duty* and *object based separation of duty*.

For example, in an office system, a manager may prepare an invoice and approve an invoice, but should not be able to approve his/her own invoice. This specification is shown below:

```

OPERATION ApproveInvoice
PRECONDITION
  #(PrepareInvoice.finish(invoker=thisUser))=0

```

Preconditions and activation constraints also help in specifying coordination policies among collaborating participants. As described in Section 1.1, coordination between the collaboration participants can be of different types. The following is an example of an inter-role coordination. In the *ChapterAuthoring* activity described in Figure 1.1, the *Editor* role has the following activation constraint.

```

ROLE Editor {
  ACTIVATION_CONSTRAINT
    #(Reviewer.StartReview.finish) = 3
}

```

This condition, along with the precondition for the *StartReview* operation of the *Reviewer* role, ensures that the editor cannot perform any operations until the chapter's contents have been reviewed independently by three reviewers. Similarly, Figure 1.4 also illustrates two forms of intra-role coordination. The precondition for the *Write* operation of the *Author* role specifies that only one author can write the contents of the chapter during one iteration.

## OPERATION Write

## PRECONDITION

```

    #(Write.start) - #(SubmitChapter.finish) = 0
    & #(SubmitChapter.start) - #(Editor.SubmitComment.finish)=0

```

This is a form of *cooperative* participation where the members of the role share the responsibilities of that role. *Independent* participation is exhibited in the *Reviewer* role's *StartReview* operation. The precondition for this operation,  `#(StartReview.start(invoker=thisUser))=0`, states that each reviewer can review the chapter contents only once. This constraint also implies that the reviewers write independent reviews. If the constraint is not based on the invoker's id, then only one of the reviewers can write a review for that chapter.

In the example of the *ChapterAuthoring* activity, the preconditions for the various operations of the *Author* role serve to provide a strict order for the sequence of operations in the activity. The preconditions and activation constraints enforce the desired task-flow of the *ChapterAuthoring* activity as described in Section 1.2.

### 1.3.4 Activity Specification

```

ACTIVITY DocumentAuthoring (OBJECTS (Document doc),
                             ASSIGNED_ROLES Author, ExternalReviewer, Supervisor) {
  OBJECT_TYPE Chapter (CODEBASE=http://codeserver) {
    METHOD writeContent {PARAM Content}
    METHOD readContent {RETURN Content}
  }
  ROLE Author {
    ADMISSION_CONSTRAINTS
    !member(thisUser, ExternalReviewer)
    OPERATION startChapterAuthoring {
      ACTION {chapter=doc.getChapter()
              new ACTIVITY ChapterAuthoring(chapter, Author=thisUser)}}
  }
  ROLE ExternalReviewer {
    ADMISSION_CONSTRAINTS
    !member(thisUser, Author)
  }
  ROLE Supervisor {
    ADMISSION_CONSTRAINTS
    #members(thisRole)<=2
  }
  ACTIVITY ChapterAuthoring {
    .....
    ACTIVITY Review { ... }
  }
}

```

Fig. 1.3 Specification of Collaborative Document Authoring Activity

An activity definition contains the specification of shared objects, roles, ownership, static assignment of users to these roles, references to objects of

```

ACTIVITY ChapterAuthoring (OWNER parentActivity.Supervisor,
    OBJECTS (DocumentAuthoring.Chapter chapter),
    ASSIGNED_ROLES Author) {
    TERMINATION_CONDITION #(Author.FinalPublish.finish) > 0
    OBJECT_TYPE Review (CODEBASE=http://codeserver) {
        METHOD writeReview {PARAM ReviewType}
        METHOD readReview {RETURN ReviewType}
    }
    OBJECT_TYPE Comment (CODEBASE=http://codeserver) {
        METHOD writeComment {PARAM CommentType}
        METHOD readComment {RETURN CommentType}
    }
    ROLE Author {
        ADMISSION_CONSTRAINTS
            member(thisUser, parentActivity.Author)
        OPERATION WriteChapter {
            PRECONDITION #(WriteChapter.start) - #(SubmitChapter.finish)=0
                & #(SubmitChapter.start) - #(Editor.SubmitComment.finish)=0}
            ACTION chapter.writeContent(content)}
        OPERATION SubmitChapter {
            PRECONDITION #(Write.finish) - #(SubmitChapter.start)>0}
        OPERATION ReadComment {
            PRECONDITION #(Editor.SubmitComment.finish)>0
            ACTION comment.readComment()}
        OPERATION FinalPublish {
            PRECONDITION #(ReadComment.start)>0}
    }
    ROLE Reviewer () {
        ADMISSION_CONSTRAINTS
            (member(thisUser, parentActivity.Author)
            & !member(thisUser, Author)
            & #(members(thisRole) ∩ members(parentActivity.Author))<1)
            ||((member(thisUser, parentActivity.ExternalReviewer)
            & #(members(thisRole) ∩ members(parentActivity.ExternalReviewer))<2))
        ACTIVATION_CONSTRAINTS
            #(Author.SubmitChapter.finish) >0
        OPERATION StartReview {
            PRECONDITION #(StartReview.start(Invoker=thisUser))=0
            ACTION { review=new OBJECT(Review)
                new ACTIVITY Review(chapter, review) }}
    }
    ROLE Editor (REFLECT parentActivity.Editor) {
        ADMISSION_CONSTRAINTS
            member(thisUser, parentActivity.Editor)
        ACTIVATION_CONSTRAINTS
            #(Reviewer.StartReview.finish)=3
        OPERATION CreateComment {
            PRECONDITION #(CreateComment)=0
            ACTION { comment=new OBJECT(Comment) }}
        OPERATION WriteComment {
            PRECONDITION #(CreateComment) > 0
            ACTION { comment.writeComment(data) }}
        OPERATION SubmitComment {
            PRECONDITION #(Author.SubmitChapter.finish) - #(SubmitComment.start) > 0
                & #(WriteComment.finish) > 0 }
    }
    ACTIVITY Review { ... }
}

```

Fig. 1.4 Specification of Chapter Authoring Activity

```

ACTIVITY Review(OWNER parentActivity.Editor,
                OBJECTS (DocumentAuthoring.Chapter chapter,
                        DocumentAuthoring.Review review)) {
  ROLE Referee {
    ADMISSION_CONSTRAINTS
      member(thisUser, parentActivityReviewer)
      & thisActivity.creator=thisUser
      & #members(thisRole)<1
    OPERATION Read {
      ACTION chapter.readContent()}
    OPERATION Write {
      ACTION review.writeReview(data)}
    OPERATION Submit {
      PRECONDITION #(Write.finish)>0}
  }
  ROLE ProofReader (REFLECT parentActivity.Editor) {
    ACTIVATION_CONSTRAINTS
      #(Referee.Submit.finish)>0
    OPERATION ReadReview {
      ACTION review.readReview()}
  }
}

```

Fig. 1.5 Specification of Review Activity

other activities, and conditions which will terminate the activity. In Figure 1.3, we describe a specification of the *DocumentAuthoring* activity as shown in Figure 1.1.

The nested *ChapterAuthoring* and *Review* activities are shown separately in Figures 1.4 and 1.5, respectively. In Figure 1.3, participants are assigned to the *Author*, *ExternalReviewer*, and *Supervisor* roles when an activity of this template is instantiated, as specified by the ASSIGNED\_ROLES tag in the activity declaration. Also, a *Document* object with multiple *Chapter* objects is passed to this activity.

Participants in the *Author* role can instantiate any number of *ChapterAuthoring* activities with *Chapter* objects. The participant, who instantiates the nested activity, is assigned to the *ChapterAuthoring* activity's *Author* role as specified. The *Supervisor* role specifies a cardinality constraint for role admission.

As shown in Figure 1.4, the owner of an instance of the *ChapterAuthoring* activity is the *Supervisor* role of the parent *DocumentAuthoring* activity. During instantiation, an object of type *Chapter* has to be passed to this activity, and members of the *Author* role of this activity have to be assigned. The definition for this activity includes the specification of two object types: *Review* and *Comment*.

The *Reviewer* role of the *ChapterAuthoring* activity has admission constraints composed of multiple conditions. The first three ensure that only one member of the parent *DocumentAuthoring* activity's *Author* role, who is not a member of the current activity's *Author* role can be in this role. The

remaining admission constraints guarantee that, only two external reviewers can join this role.

When the author of a chapter completes a draft of the chapter’s content, it is published to the chapter’s reviewer and editor by invoking the *SubmitChapter* operation. This results in the coordination actions of making the chapter’s content available to the participants in the *Reviewer* and the *Editor* roles. This operation also enables a reviewer to write a review of the chapter. The reviewer of the chapter cannot compose the review until the chapter content has been written and submitted. Similarly, the editor cannot write the comments until all reviewers have reviewed the contents of the chapter independently. After the editor publishes the comments, the author may modify the chapter contents and either publish the final version or simply submit the chapter again. In this case, no new review activities will take place, only the editor will be able to publish new comments. This cycle goes on until the author invokes the *FinalPublish* operation, after which the contents of the chapter are not editable and the *ChapterAuthoring* activity terminates.

This example shows how coordination can be specified in nested activities and how shared objects move among these activities.

### 1.4 MIDDLEWARE EXECUTION MODEL

Here, we briefly present the middleware execution model to discuss how policy modules are derived and enforced. The middleware provides services for role management, object management, naming, and secure communication of events. A detail discussion of the execution model is presented in [23].

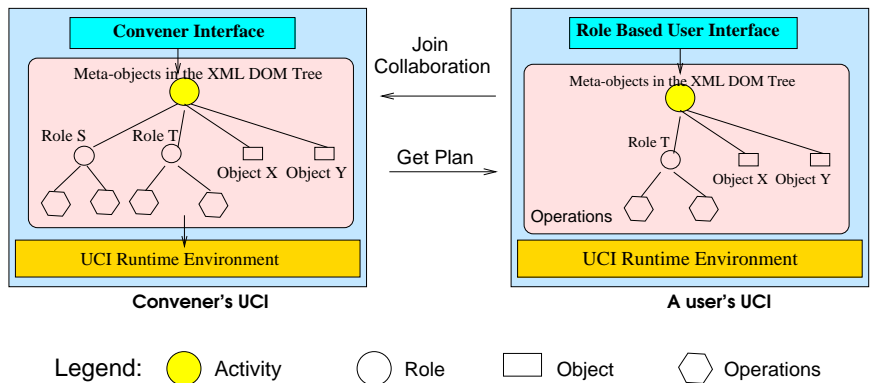


Fig. 1.6 System level view of a distributed collaboration environment

Initially, policy for a collaboration environment is specified using an activity template by the *Convener*. From the activity template, a tree structure is created that represents the various entities in the activity. This structure is

built from the XML specification of the collaborative activity and contains only the static definition of the activity. This tree structure is essentially an XML Document Object Model (DOM tree). Figure 1.6 shows the typical structure for realizing a distributed collaboration using a generic coordination facility. As activities are created at runtime, an instance tree is built on the lines of the earlier DOM structure, to represent the distributed runtime environment. Each node in the tree represents an activity, role or an object. Every participant in the system obtains a subset of this tree based on the access control policies. This subset is referred to as the *role specific view* and is accessible to each participant through his/her *User Coordination Interface* (or UCI).

Access control modules are derived from the activity template which specifies the privileges that various roles have on shared objects. In other words, for each object type within the activity, there is a template that contains a skeleton of its access control policies. When these objects get instantiated, the access control entry for the newly created objects get filled in with object instance ids and role ids that are currently active in the activity and are relevant to the objects.

With our specification model, coordination between the participants in a collaboration system is specified in terms of events and event counts. A secure distributed event service, which is a part of the middleware services, derives the subscription-notification relationships between the participants and their operations, from the specification of the activity. Before any activity is created, this subscription-notification model is based on the templates of the various activities, roles and objects that may be created at runtime. As these entities are instantiated at execution time, the parameterized values in templates get bound and the policy modules are accordingly updated.

## 1.5 RELATED WORK

Our work is similar to COCA [12] and DCWPL [2] in their approach of constructing a distributed collaboration environment from a high level specification. COCA [12] is a logic-based coordination policy specification language for interactive CSCW applications which views security policy as an integral part of coordination policy. At the implementation level, COCA is strongly tied with IP multicast models and Prolog which may not scale for a large collaboration. DCWPL [2] addresses user level mechanism to deal with group interaction issues and is limited to its predefined policies and functions. Neither of these approaches support active security policies, like policy for dynamic separation-of-duty constraints. These approaches can only specify a coordination policy based on previously executed operation and cannot independently specify discretionary access control for an object. Theoretical work for specification of confidentiality in CSCW systems using  $Z$  notation is examined in

[7], which is limited to its theoretical foundation and lacks implementation model for security architecture.

The concept of role has been used in CSCW systems such as XCP [19], MPCAL [8], Quilt [11]. These applications use roles to represent groups of users with different tasks within a collaboration. They do not satisfy the security requirements specified in Section 1.1. Suite [4] presents access-control model for multiuser interfaces, mainly for coordination of shared editing-based synchronous collaboration. For that, it deals with a wide-range of access rights on shared data. In contrast, the focus of our work is mainly on a participant's role-based access to shared data, addressing the needs of data confidentiality, integrity, and dynamic security policy. In Intermezzo [5], which is designed for user-presence awareness environments, roles are dynamic groups of users with whom access control policies are associated. The use of roles in a collaborative software development environment is presented in [3]. Both of these [5, 3] lack specification of tasks associated with user roles and policies for their coordination.

A task-based constraint specification language for workflow management systems is discussed in [1]. There, constraints are specified with a mapping between roles and tasks. However, it is not clear how such high level specifications of collaborative activities are realized in an implementation. In contrast, our work specifies both the security and the coordination policies, with realization of such policies in an implemented system. SecureFlow [10] imposes workflow authorization constraints on tasks using Authorization Template(AT), which is a tuple specifying privileges to be granted to a subject of a given role on a object of a given type during a given time interval. There, the permissions are activated based on tasks. In contrast, an activity in our model has multiple roles and object types, and the interaction among these roles and objects through operations. If one thinks AT as a method specification, an activity is a module specification with multiple methods. An activity specification may contain multiple tasks or operations and is able to capture workflow stages. For ease of policy specification in distributed systems, the domain concept is used in [24]. However, their domain is a grouping of various objects based on physical location, types, responsibility etc. for the convenience of a policy manager. An activity specification defines a protection domain for several types of objects and roles based on the interaction among these objects and roles.

In [20], team based access control (TMAC) is presented as an approach of applying RBAC in workflow like collaborative environments. In conjunction with RBAC, TMAC proposes an active security model similar to trigger oriented active databases. A team has participants from different roles. Our concept of activity, where roles are created or reflected inside an instance of an activity, subsumes the team concept.

A framework for role based access control (RBAC) models with constraints and role hierarchies is presented in [16]. In [13], role based management (RBM) is presented, which incorporates role obligations, i.e, actions which

must be performed when certain events occur, and it views role definitions similar to classes, which can be instantiated or reused through inheritance. In our system, roles are defined as instantiable classes in the context of an activity definition. Instances of the same role in different instances of an activity represent different protection domains. Our role admission criteria are conceptually similar to those presented in the context of RDL (Role Definition Language) in [9]. The focus of that work is on distributed and decentralized management of roles in a distributed service model. In our model, we are concerned with an integrated centralized specification of collaboration environments. Our goal is also to derive an implementation from a specification. We introduce here, the concept role reflection which provides similar functionality of role inheritance and reusability as addressed in [13].

## 1.6 CONCLUSION

We have presented in this paper a coordination model for secure collaboration systems. We use this model in a policy-driven middleware for supporting rapid construction of a collaboration environment from its specifications. The specification model developed in the context of this work supports nested activities and role based coordination and security policies. In our approach, policy modules are derived from the specifications of a collaboration activity. The policy modules are related to the management of activities, roles, and objects in the collaboration. These policy modules are distributed to different users' computing sites based on the user's role and the security requirements. We have presented our specification model using a detailed example of a document authoring activity.

**Acknowledgements:** This work was supported by NSF grant ITR 0082215 and EIA 9818338. The authors are with the Department of Computer Science, University of Minnesota, Minneapolis.

## REFERENCES

1. E. Bertino, E. Ferrari, and V. Atluri. A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. In *ACM Workshop on Role-based Access Control*, pages 1–12, 1997.
2. Mauricio Corts and Prateek Mishra. DCWPL: A Programming Language for describing Collaborative Work. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 21 – 29, November 1996.

3. S.A. Demurjian, T.C. Ting, and B. Thuraisingham. User-Role Based Security for Collaborative Computing Environments. *Multimedia Review*, 4(2):40–47, Summer 1993.
4. Prasun Dewan and HongHai Shen. Access Control for Collaborative Environments. In *Proceedings of the ACM CSCW'92*, pages 51–58, 1992.
5. W. Keith Edwards. Policies and Roles in Collaborative Applications. In *Proceedings of CSCW'96*, pages 11–20, 1996.
6. Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: Some Issues and Experiences. *Communication of ACM*, 34(1):39 – 58, January 1991.
7. S. Foley and J. Jacob. Specifying Security for Computer Supported Collaborative Computing. *Journal of Computer Security*, 3(4):233–253, 1995.
8. Irene Greif and Sunil Sarin. Data Sharing in Group Work. *ACM Transactions on Information Systems*, 5(2):187–211, 1987.
9. R. Hayton, J. Bacon, and K. Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, pages 3 –14, 1998.
10. Wei-Kuang Huang and Vijayalakshmi Atluri. SecureFlow: A Secure Web-enabled Workflow Management System. In *ACM Workshop on Role-based Access Control*, pages 83 – 94, 1999.
11. M.D.P. Leland, R.S. Fish, and R.E. Kraut. Collaborative Document Production using Quilt. In *Proceedings of CSCW'88*, pages 206–215, 1988.
12. Du Li and Richard Muntz. COCA: Collaborative Objects Coordination Architecture. In *Proceedings of CSCW'98*, pages 179–188, 1998.
13. Emil C. Lupu and Morris Sloman. Reconciling Role-Based Management and Role-Based Access Control. In *ACM workshop on Role-based Access Control*, pages 135–141, 1997.
14. Matunda Nyanchama and Sylvia Osborn. The Role Graph Model and Conflict of Interest. *ACM Transaction on Information System Security*, 2(1):3–33, February 1999.
15. P. Roberts and J-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proc. of IFIP Congress*, pages 981–986, 1977.
16. Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

17. Ravi S. Sandhu. Transaction Control Expressions for Separation of Duties. In *Fourth Annual Computer Security Application Conference*, pages 282–286, December 1988.
18. R.T. Simon and M.E. Zurko. Separation of Duty in Role-based Environments. In *10th Computer Security Foundations Workshop*, pages 183–194, 1997.
19. Suzanne Sluizer and Paul M. Cashman. XCP: An Experimental Tool for Managing Cooperative Activity. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, pages 251 – 258, 1985.
20. Roshan K. Thomas. Team-based Access Control (TMAC): A Primitive for applying Role-based Access Controls in Collaborative Environments. In *ACM Workshop on Role-based Access Control*, pages 13 – 19, 1997.
21. Jonathon E. Tidswell and Trent Jaeger. Integrated Constraints and Inheritance in DTAC. In *ACM Workshop on Role-based Access Control*, pages 93 – 102, July 2000.
22. Jonathan Trevor, Tom Rodden, and John Mariani. The Use of Adapters to support Cooperative Sharing. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 219 – 230, 1994.
23. Anand Tripathi, Tanvir Ahmed, Richa Kumar, and Shremattie Jaman. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proceedings of the International Conference on Distributed Computing Systems*, 2002.
24. N Yalelis, E Lupu, and M Sloman. Role-based security for distributed object systems. In *Proceedings of the 5th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 80–85. IEEE Computing Society, 1996.