POLICY-BASED DESIGN OF SECURE DISTRIBUTED COLLABORATION SYSTEMS

a dissertation
submitted to the faculty of the graduate school
of the University of Minnesota
by

Tanvir Ahmed

in partial fulfillment of the requirements
for the degree of
doctor of philosophy

Anand R. Tripathi, Adviser

November 2004

University of Minnesota

This is to certify that I have examined this copy of a doctoral thesis by

Tanvir Ahmed

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

Date

Graduate School

Abstract

CSCW (Computer Supported Cooperative Work) systems range from traditional office automation workflow, interactive groupware, to Internet-wide virtual organizations. Contemporary CSCW systems may need to span across different organizations and security domains. In these environments, no single site or user may be trusted to enforce all the policies of a distributed CSCW system. Moreover, users' coordination and security requirements in such systems need to adapt with changes in administrative policies and user experiences.

This dissertation presents a framework for construction of secure distributed CSCW systems from their high-level specifications containing security and coordination policies. The research presented in this thesis contributes in three areas: (1) a role-based specification model for coordination and security policies, including administrative security policies, in distributed CSCW systems; (2) a verification methodology based on model checking to ensure that a specification satisfies security requirements related to availability, confidentiality, integrity, and access leakage; and (3) a design of a policy-driven middleware for constructing the secure runtime environment for a distributed CSCW system from its specification.

The role-based collaboration specification model developed separates the security and coordination concerns from the implementation of shared applications and resources. This specification model can express a wide range of coordination and security requirements, such as role admission constraints, intra- and inter role coordination, hierarchical structuring of activities, various forms of "separation of duties",

confidentiality, context sensitive access control policies, and meta-level administrative security policies. Finite-state based model checking, using the model checker SPIN, is utilized for verification of security requirements. The challenges in managing the complexity of finite state based verification for collaboration policies are addressed, and a verification methodology based on aspect specific verification models is developed.

The policy driven middleware that we have developed as a proof-of-concept automates the realization of a distributed CSCW system from its specification integrating shared applications and resources. This dissertation addresses challenges in designing the policy driven middleware, such as selection of secure sites for policy enforcement, policy module derivation and distribution, secure event service, and session management. We have evaluated our approach by building several experimental collaborative applications with various coordination and security policies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this dissertation, a framework for specification, verification, and construction of secure distributed collaboration systems is developed. The framework includes:

1. A methodology for specifying and verifying security and coordination requirements of collaboration systems; and

2. An architecture of a policy driven middleware to construct a secure distributed collaboration system, integrating shared applications and resources, based on its specification.

1.1. Computer-Supported Cooperative Work (CSCW)

The examples of computer supported collaboration include online conferencing, product design and development, authoring of documents, workflow in an office environment, healthcare activities in a hospital, and collaboration among different organizations. In Figure 1.1[1], historical perspective of various systems for computer supported collaboration are presented. Early examples of collaboration systems are data processing systems, decision support systems, and management information systems that are responsible for organization level management aspects (Grudin and Poltrock, 1997). Later, systems for software development, collaborative design, office

[1]The time line is based on (Grudin and Poltrock, 1997; Hollingsworth, 2004)

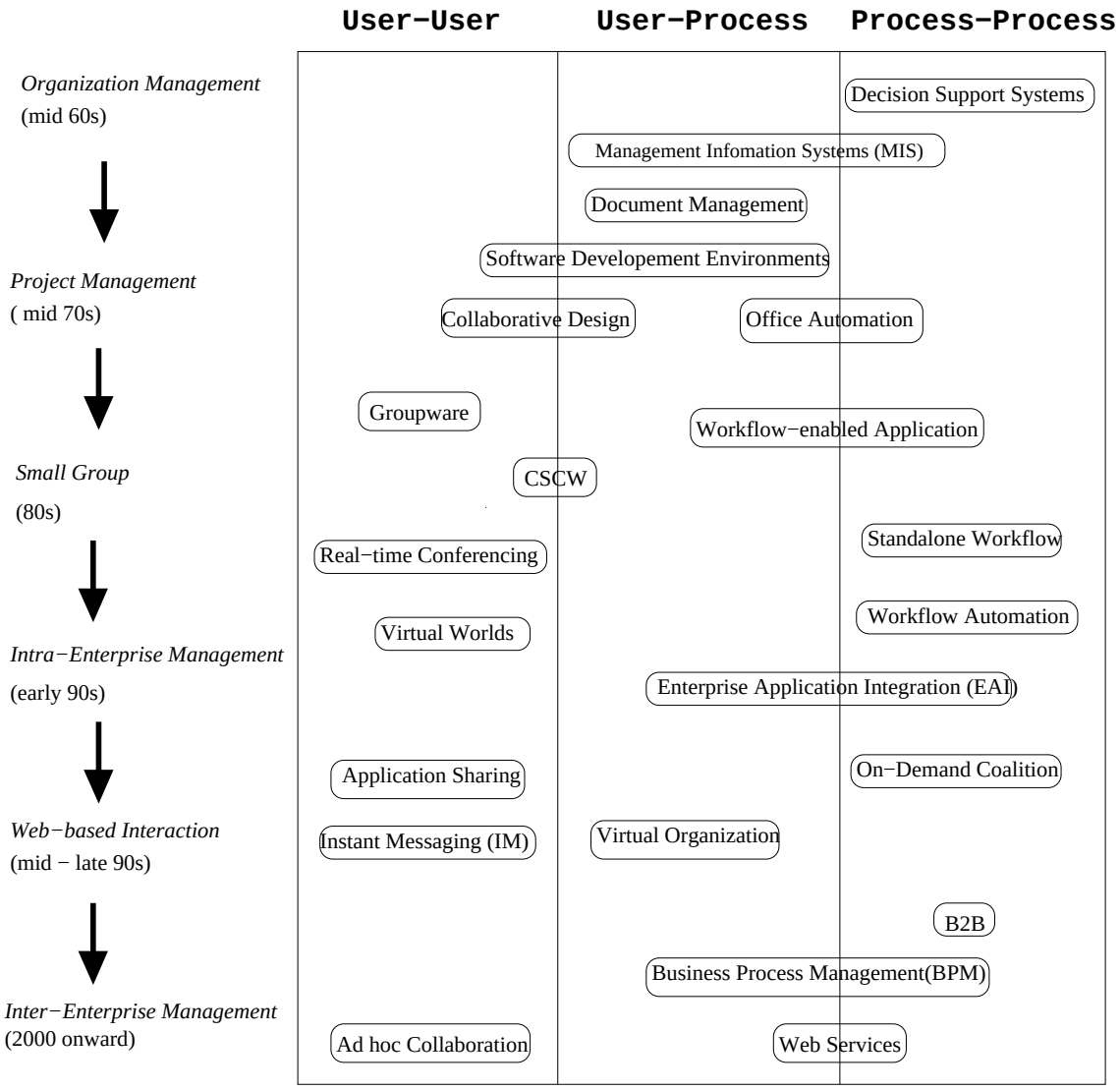

Figure 1.1. Time line of CSCW and workflow systems: management aspects and interaction models

automation, document management, and other systems that manage projects involving multiple users were introduced. Many of these systems are targeted towards small group activities and are termed *groupware*. Groupware systems are differentiated from other multi-user systems, such as database systems, based on their application level requirements of data sharing (Greif and Sarin, 1987; Grudin and Poltrock, 1997). Compared to concurrent access of data through transaction management facilities in database systems, groupware systems coordinate interacting users, where the interactions can occur in real-time. These interactions in groupware resemble long term transactions. Moreover, in contrast to traditional transactions in database systems, during the interactions in groupware systems, the users are usually aware of eachother's presence or other user-level contextual information, hence these systems are often termed as *group-aware*. During 80's, the term Computer-Supported Cooperative Work (CSCW) (Greif and Sarin, 1987) was introduced for group-oriented systems; however, CSCW is a multidisciplinary field that addresses all aspects of group activities including ethnographic, social, technological, and theoretical issues for enabling group activities.

Formally, in groupware systems, multiple users cooperate using shared data and applications towards some common objectives (Ellis et al., 1991). It is envisioned in CSCW systems that the collaborators may not initialy know their common objectives, but rather "discover" similar goals after their interactions have progressed to a certain point (Oravec, 1996). Due to enabling facilities to discover common grounds with other cooperating users, flexibility of sharing information including meta and contextual information is viewed as a primary attribute of groupware systems. CSCW researchers have differentiated cooperative work from coordinated work based on that in the cooperative work users share the work objective, which enables the cooperating users to adapt their actions with each other to achieve the objective (Bardram, 1998).

On the other hand, workflow systems were introduced following the tradition of office automation systems that define and automate tasks in office environments. Workflow automation systems are derived from existing and conventional practices

that consist of a set of well-defined activities to execute some enterprise-wide process. These systems are criticized for their emphasis on workflow processes rather than users (Hollingsworth, 2004). For commercial use, many groupware applications become workflow enabled, i.e., groupware applications became aware of workflow stages.

Enterprise Application Integration (EAI) is an enactment of workflow systems for intra-enterprise management. Current workflow systems are synonymous with Business Process Management (BPM), which also incorporates inter-organizational workflow or B2B (Business-to-Business). *On-demand coalition* is another type of cross organizational workflow system. Example coalition systems are disaster reliefs and war-time sharing of data. In the commercial arena, E-Services are the outgrowth of E-Commerce or the web based interaction of buyers and sellers (Stafford, 2003). E-Services model has evolved to Web Services that provide web-delivered services (W3C, 2002b). The motivation behind Web Services is lightweight integration of business processes supporting interoperability across platforms or technologies. A primary attribute of Web Services is XML technologies that define, describe, discover, and invoke these web-delivered services.

In Figure 1.1, different CSCW and workflow systems are presented across three areas: groupware that coordinate users, workflow enabled applications that coordinate users with workflow processes, and workflow systems that coordinate processes of different applications or organizations. Distributed workflow systems, such as BPM and Web Services, support interactions of inter-organizational processes and interactions of users in these processes. On the other hand, systems such as online interactive games represent distributed groupware that support interactions of users situated in different locations. Due to popularity of mobile devices, ad hoc collaboration systems to share network resources are emerging.

In Figure 1.2, a user-process matrix classifies CSCW and workflow systems based on their emphasis on supporting user or process interactions. The top two cells representing groupware and workflow-enabled applications are classified as CSCW systems

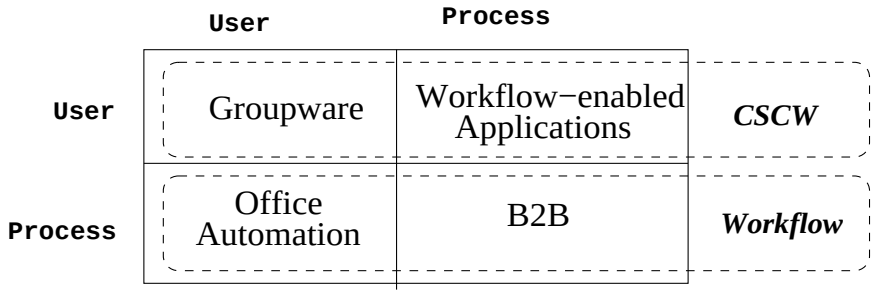

Figure 1.2. User-process matrix for CSCW and workflow systems

as their emphasis is on supporting group works. The bottom two cells represent workflow systems as their emphasis is on process automation of organizational activities.

To categorize the wide range of CSCW systems, several CSCW researchers, such as (Grudin and Poltrock, 1997; Dourish, 1999), have looked into the types of technologies that collaboration systems support: (1) communication, (2) information manipulation and storage, and (3) coordination (See Figure 1.3). Most of the collaboration systems do not exclusively fall into a single category, but rather incline toward one of these supporting technologies. Systems such as electronic mail, real-time conferencing, multicast video and audio are usually associated with communication technology. Examples of shared information space technology include shared whiteboard, application sharing, meeting facilitation, virtual worlds, threaded discussions, document management, and information management, e.g., Lotus Notes (Orlikowski, 1992). Lastly, examples of coordination technology are calendars, scheduling, and office automation systems. All CSCW systems support some form of coordination technology.

1.2. Coordination in CSCW Systems

The primary functionality of collaboration systems is to coordinate cooperating users. Coordination is the process of managing dependencies among users' work activities (Malone and Crowston, 1994). In a broader term, coordination process

Communication	Information Manipulation	Coordination
Electronic Mail	Shared Whiteboard	Calendars
Real−time Conferencing	Application Sharing	Scheduling
Multicast Video/Audio	Virtual Worlds	Office Automation

Figure 1.3. Example CSCW systems based on types of supporting technologies

manages flow of data, sharing of resources, synchronization of resource usage, and other relations among work activities, such as decomposition of work activities into tasks/subtasks and planning how such tasks will be performed (Malone and Crowston, 1994).

The nature of coordination can range from tightly coupled synchronous interactions, as in online conferencing and shared whiteboard environments, to loosely coupled asynchronous interactions, which are largely characterized by workflow environments. Workflow management systems usually consist of a set of well-defined activities to execute some enterprise-wide process. In real-time synchronous collaborations, users are connected to the system simultaneously and their interactions occur through interactive sharing and manipulation of graphical and multimedia objects. Generally, the interactions tend to be unstructured, often spontaneous, and the coordination management is concerned with concurrency control for access to shared resources. In contrast, in loosely coupled collaboration, all the users may not be present at the same time, and their coordination actions tend to be coarse-grain. These are usually workflow-enabled systems. These workflow like systems represent a form of collaboration where the objective is to support the operational activities of an organization. A large workflow environment occasionally have instances of tightly coupled interactions among its participants.

1.3. Motivation

Web has become an integrated part of current collaboration environments. On the other hand, the enabling open network infrastructure has raised security concerns for every day applications as oppose to security concerns in traditional government or commercial systems. It was widely accepted, as expressed by Clark and Wilson (1987), that government or military systems are mainly concerned with confidentiality or information flow related security properties; On the other hand, commercial systems tend to emphasize on integrity and availability of data and services. With the unsecure network, the security attributes – namely integrity, confidentiality, and availability – desired for a system are not limited to a specific attribute. Many of these systems interact with users that are strangers or reside in different organizations or countries. This itself requires reexamination of the traditional security models that are used to express and enforce security policies.

Social aspects for usage of these Internet-wide collaboration systems have raised demand for security services that were not prevalent earlier. For example, due to recent privacy concerns and laws in healthcare information systems, confidentiality has become a primary concern for many commercial systems. Utilizing the Internet, these systems share data crossing their organizational and security boundaries. Computations in these systems are increasingly performed across organizational boundaries, often close to data. In addition, the open network has introduced new security threats, such as *denial of service*, *snooping* for unauthorized interception of information in the network, *spoofing* or *masquerading* based on *stolen credentials* among other techniques, and traditional attacks on the protection mechanisms using new forms of *Viruses*, *Worms*, and *Trojan Horses*. Security models are developed based on assumptions on system environments including security threats. The new threats require consideration for security models that support security policies in open environments.

The dissertation topic presented in this thesis is motivated by the following attributes of current collaboration systems and their unique security requirements.

1.3.1. Attributes of Contemporary Collaboration Systems

Current collaborative systems have several challenging attributes, which have motivated the research presented in this dissertation:

1. [Collaboration systems are evolving and emerging:] With advances in technology new devices, tools, and application tools are constantly integrated into a collaboration environment. This warrants new collaboration systems or reconfigurations of existing systems. Ackerman (2000) and Grudin (1994) have identified dynamic social aspects that influence group activities as a challenge for CSCW system development. A primary characteristics of collaboration systems is that they need to adapt with new tools or artifacts, changes in administrative policies, and user experiences.

2. [Reconstruction of coordination policies:] According to Bardram (1998), coordination policies in collaboration systems reflect the means of cooperation, where cooperation is the reflection on the object of work. Cooperation is established through well-established coordination process. Due to changes in motivations of group activities or directions of organizations, the cooperation process may require re-examination. In some cases, the work objective itself has to be re-examined or re-conceptualized for corrective adjustments. Such adjustments result in restructuring the coordination process. The ability to re-organize coordination processes and to realize new coordination processes is identified as one of the primary requirements for the success of collaboration systems (Suchman, 1994; Winograd, 1994). From a decentralized perspective of CSCW systems, Carla Simone (1995) and Schmidt and Simone (1996) emphasized on adaptable coordination mechanisms for articulation of distributed activities.

3. [Span across organizations and security domains:] In Internet-wide collaboration, users and shared objects are inherently distributed, and users from different organizations or administrative domains need to cooperate. There are several motivations for decentralized management of collaboration systems. Other than the well-known motivations of scalability and elimination of "single point of attack", collaboration systems need decentralized management as no single organization, site, or user could be trusted with the management of all the aspects of these systems.

4. [Decentralized management of collaboration activities:] With the popularity of peer management of network resources, many recent collaboration activities are peer-to-peer, such as file sharing. With increasing concern in security and privacy, users may prefer their own computing devices for managing resources they share during a collaboration.

5. [Formed by ad hoc integration of users and shared resources:] Recently, cross organizational collaboration are identified with Web-Services. Web services address cross-organizational workflow orchestration involving agreement among participating organizations based on pre-defined specification. This usually involves delegation of required services to distributed service providers. On the other hand, distributed CSCW systems require delegation of coordination processes based on pre-defined specifications integrating shared resources and distributed users.

1.3.2. Execution Environment for Distributed Collaboration

Figure 1.4 shows a typical example of the execution environment for distributed collaboration systems. In this figure, a typical collaboration activity, which spans across three security domains, is shown. In this model, a security domain can be a predefined group of users and resources representing an administrative or organizational domain. Within a domain, users and resources are managed under the same administrative entity. In the figure, users from these domains participate in a collaboration sharing resources. The collaboration activity may have nested sub-activities. In the figure, two sub-activities are shown, one of which only involves a subset of

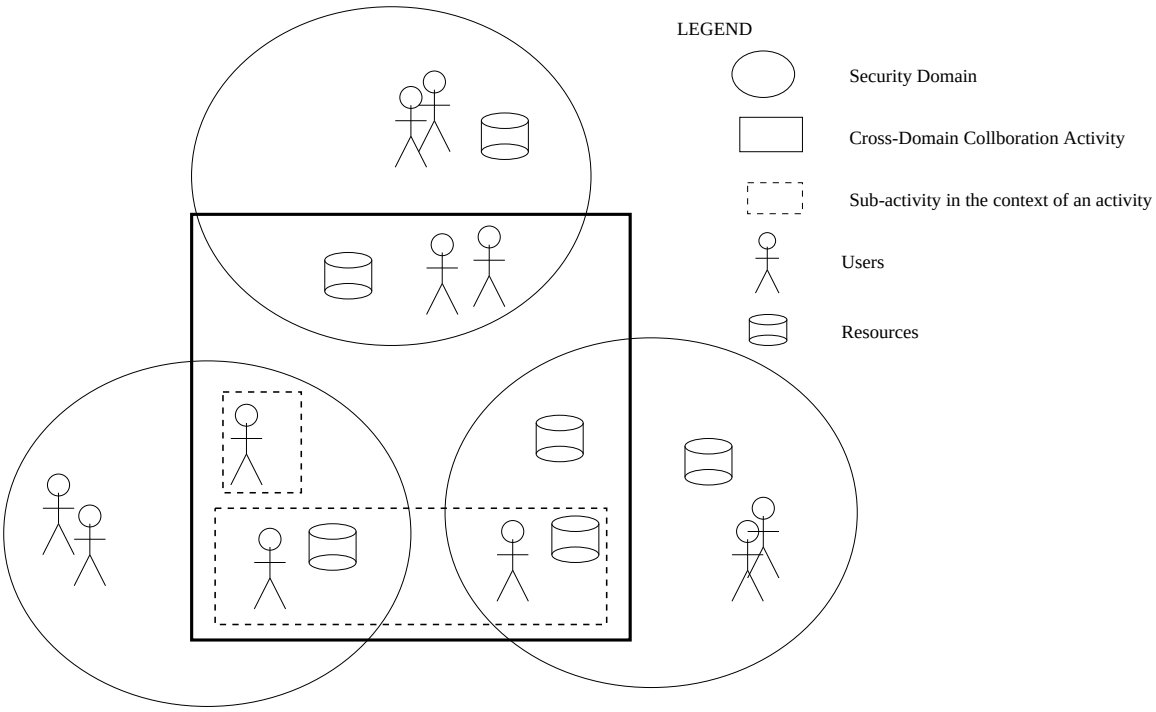

Figure 1.4. Execution environment for decentralized collaboration systems

the participating domains and the other sub-activity is contained within a domain. Within a domain, users may participate in collaboration activities and store resources in nodes under their control, such as personal computer, or in servers trusted by the users in the domain.

1.3.3. Trust in Decentralized Administration

For runtime enforcement of policies, we need to ensure that policies related to role membership management, object access, task creation, task coordination, and other administrative aspects of managing collaborative activities are enforced in nodes managed by entities who are *trusted* by other concerned participants of the collaboration. Before, we formally define what we mean by administrative trust for collaboration systems, the term "trust" is evaluated based on existing publications in the computing literature.

Policy enforcement mechanism plays an important role in developing security policy specification models. Without proper authentication, access control models are useless. If a user can acquire multiple identities within a domain, various cross-organization level security policies, such as Chinese Wall policy, cannot be enforced. For the same reasons, if the enforcement mechanisms are under control of an administrator who cannot be trusted, security policies cannot be enforced.

In the paradigm of trusted computer system initiated by the *Trusted Computing System Evaluation Criteria* (TCSEC), trust is viewed as a property of a system (Denning, 1993; Dobry and Schanken, 1994). According to the Criteria, the Trusted Computing Base (TCB) is the only part of the system that needs to be evaluated to prove that trust is present. Hence, the trust on the TCB can be viewed as the knowledge acquired based on derived trust from: (1) the trust in the formal verification methods and (2) the trust in the trust derivation mechanism (Jsang, 1996). Later TCB concept is extended in distributed TCB, where untrusted communication medium is introduced under TCB boundaries, as oppose to traditional untrusted

applications run over a TCB boundary, and messages through untrusted media is proposed to be protected through cryptographic techniques (Casey et al., 1988).

Trust is defined as the result of an assessment made by an observer about a person, organization, or any other entity (Denning, 1993). In distributed systems, trust is classified where being trusted in a class means that an entity is trusted to perform specific task, such as providing identification for another entity, providing good quality keys, maintaining secret, and providing options about the trustworthiness characteristics of other entities (Yahalom et al., 1993). As opposed to trust derived in TCB from the proof of the specification of functionality, in recent Internet-wide systems, the trust concern in shifted to the interacting entities including human users and services. Trust is discussed in the context of certification of an entity's attributes, such as roles. These attribute certificates are also called credentials. Often these credentials are cascaded, i.e, validity of a credential depends on the validity of other credentials. Cost-effective solutions to find the right set of credentials for an authorization and to revoke credentials are challenges of trust management systems.

In collaboration environments, in addition to be concerned about the trust on the enforcement mechanisms and the credential management, research challenges include ensuring trust on entities with management responsibilities. In decentralized setting, this implies assigning trust on an entity for administrative task of enforcing policies and performing management and obligatory functionalities that arise at runtime.

1.3.4. Security Policies in Collaboration Systems

Security policies are, in practice, concerned with subjects' access to resources under specified conditions (Ryan et al., 2001). In Chapter 2, we closely investigate unique security requirements present in current collaboration systems. However, we have identified and emphasized on the following two aspects of security requirements in collaboration systems:

1. [Context-Sensitive Security Policies:] Policies related to security attributes – privacy or confidentiality, integrity, and resource access – in CSCW need to handle various context-sensitive aspects of the collaboration environment. Such context-sensitive security constraints can be based on the coordination state of the cooperating users. In collaboration systems, coordination constraints are often weaved with access control concerns. For example, in an examination activity, students can only view the question after the examiner has released it and only during the exam-session. Although this resembles a coordination concern, it is also an access control constraint. Such examples not only show that *there is a strong tie between security and coordination policies, it also emphasizes that security and coordination policies need to be in concert for a collaboration system.*

2. [Decentralized Administration:] Without a proper or "trusted" assignment of users in *administrative roles* (Sandhu et al., 1999), participants in an administrative role may try to acquire extended privileges by violating global security requirements. In decentralized management, *protection mechanisms* enforcing the policies may be under the control of different administrators. An administrator may not enforce the part of the policy it is entrusted with, thus possibly resulting in violation of overall collaboration policies. Moreover, these untrusted users in administrative roles may deliberately try to violate sensitive security requirements by manipulating the policies under their control.

1.4. Objective

The primary objective of this thesis is to develop techniques for constructing secure distributed collaboration environments from their high level specifications capturing coordination and security requirements. This approach decouples coordination and security aspects of a collaboration from the implementation of the functionalities of shared objects or application-level components. Our goal is to construct collaboration environments with different coordination and security policies with the same set of objects and software resources for implementing collaborative tasks. Similar challenges are addressed by component integration frameworks for implementing distributed collaboration systems (Banavar et al., 1998; Basin et al., 2003); however, these integration specifications do not separate the coordination and security concerns.

In Figure 1.5, the goal of our policy-driven approach of building collaboration environments is shown. Based on different sets of coordination and security policies, the policy-driven middleware integrates the same shared applications and resources to build different collaboration environments. This approach enables the cooperating users to adapt a collaboration specification, which fulfills their coordination and security requirements. Moreover, the specification can be changed to reflect new coordination or security requirements, and new specification can be designed for a new set of requirements. The specification of the requirements is termed *policies*. A collaboration specification contains three distinct types of policies: coordination, access control, and meta-level administrative policies.

In Figure 1.6, the life-cycle of our of collaboration systems is shown. At Step 1, designers generate a collaboration specification based on a given set of shared objects and required coordination and security policies. Many such collaboration specifications can be designed and stored, as shown in Step 2. At runtime (Step 3), users select a collaboration specification based on desired security and coordination requirements and realize a collaboration environment.

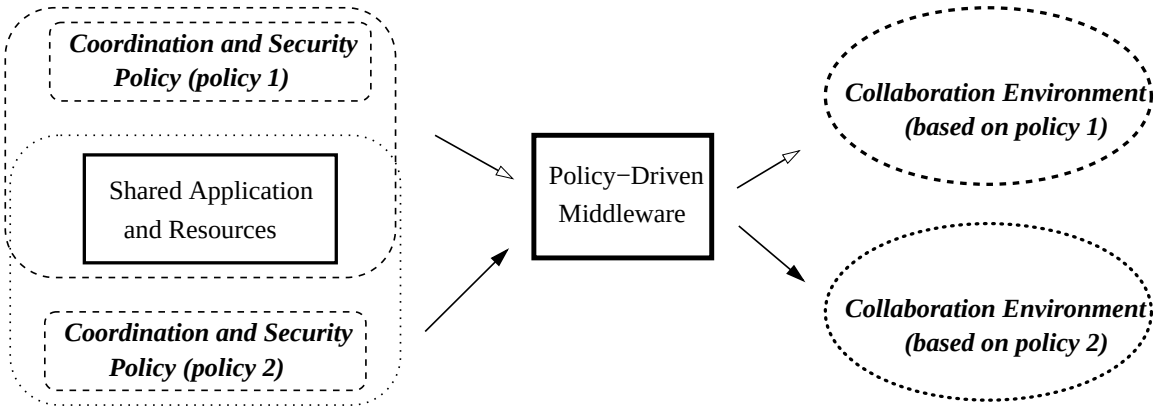

Figure 1.5. Construction of different CSCW environments based on different coordination and security policies

This research objective requires a formal specification model for security and coordination policies, and development of a verification process to ensure that the requirements can be correctly enforced given a specification. Security policies specify the conditions that must be satisfied when subjects access resources. Traditional access control models, such as access matrix (Harrison et al., 1976) or *take-grant* model (Snyder, 1981), cannot specify such conditions.

In collaboration systems, role based models (Sandhu et al., 1996; Ting, 1988; Greif and Sarin, 1987; Lupu and Sloman, 1997; Bacon et al., 2002) for coordination and security policy specification gained popularity for their ability to specify policies without knowing the identities of the users who will participate at runtime. A role defines a set of privileges or access-rights. In contrast, a user group is defined to aggregate a set of users for ease of policy specification: however, the identities of the users are usually known before the grouping. At anytime, a user has all the privileges derived from his/her memberships in different roles. In contrast, a primary advantage of role based models is that relations or constraints can be specified among roles, so that

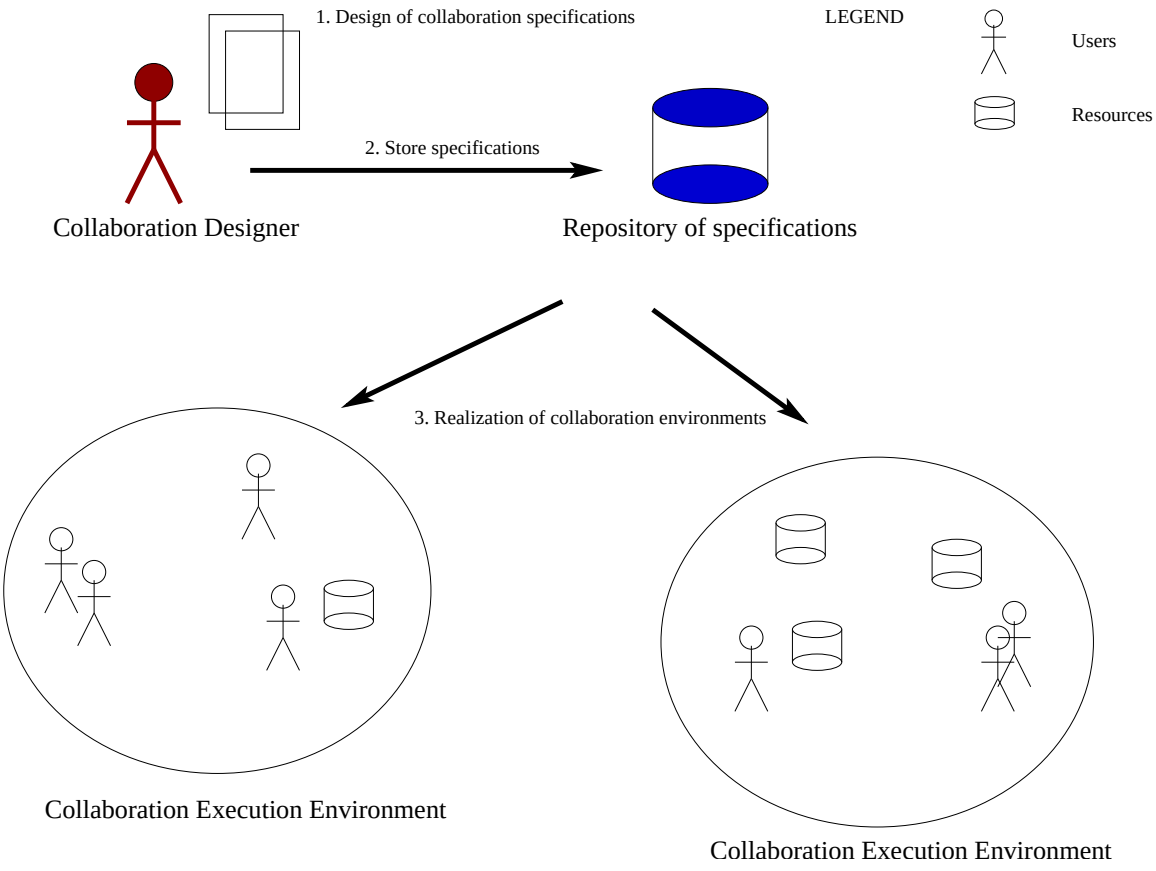

Figure 1.6. Life-cycle of decentralized collaboration systems

only subsets of the roles assigned to a user can be activated at anytime. In existing literature, a role represents a set of privileges, specific task competency, specific duty assignment, authorization, responsibilities, and capabilities (Sandhu et al., 1996; Baldwin, 1990).

1.5. Contributions

The goal of our research is to realize distributed CSCW systems from their high level specifications. The primary contribution of this dissertation is twofold.

1. *A methodology for security and coordination policy specification for distributed collaboration. This includes:*

   (a) *A role based specification model for CSCW systems.*

   (b) *Verification of coordination and security requirements.*

2. *Design of a middleware for policy driven construction of secure distributed collaboration.*

1.5.1. A Role based Model for Security and Coordination Policies

As part of this dissertation, a role based model to specify security and coordination policies in distributed collaboration systems is developed. The model is developed based on unique context sensitive security requirements that are present in current distributed collaboration systems. The role model developed facilitates specification of administrative level security requirements in specifying security policies for distributed collaboration systems. The deficiencies in existing role based security models are discussed in Chapter 3

1.5.2. Verification of Collaboration Specifications

The policy specification methodology includes development of a verification model to ensure that the specification satisfies security requirements (Gunter et al., 2000). The focus of the verification process is on a methodology for security policy specification to ensure that sensitive security requirements cannot be violated by untrusted participants in distributed management of collaboration systems. The objective of the verification process is to ensure that the specified constraints do not violate any desired coordination and security properties, such as:

- User interactions follow coordination and task-flow requirements;

- Roles do not have conflicting or inconsistent constraints;
- Confidential information cannot flow to unauthorized users;
- Authorized information can be accessed;
- Any temporal or conditional constraints on accessing objects can be satisfied;
- No rights can be leaked to unauthorized users.

In our verification approach, some of the distributed security domains can be under the control of administrators who may not be trusted. The behavior of the untrusted administrators is modeled to verify security properties. The solution proposed for administrative policy specification is based on the fact that in distributed collaboration environments, a participating user may not be trusted in regard to management of all the aspects of a collaboration; however, the same participant may be trusted to manage some specific aspects or entities of a collaboration. In our specification methodology, the problem is related to identifying the roles that cannot be trusted to enforce policies related to specific concerns of a specification and determining *safe* assignments of administrative roles.

The approach taken in the thesis, is to model administrative roles, which are trusted entities to manage different collaboration aspects or entities, such as role, objects, and activity management. However, modeling of such trusted entities has several challenges:

1. Find participants who are trusted to be assigned in an administrative role for specific management aspect.
2. Change the administrative role, as collaboration progresses and the role initially trusted cannot be trusted any longer for specific management aspect.
3. Specify facts about trusting participants for specific management aspects so that the specification can be verified against the security requirements. As such listing can be exhaustive, only the facts that are required for a verification are specified.

In this thesis, as part of the verification methodology, we present how finite-state techniques, such as model checking, can be utilized for correctness verification in a role based collaboration model. A motivation of our approach is to use an existing model checking tool, SPIN (Holzmann, 2003), to verify security requirements of distributed CSCW systems during the design phase.

1.5.3. Design of a Policy-Based Distributed Middleware

In our approach, a collaboration environment will be realized in steps as discussed below. These steps are also shown in Figure 1.7.

1. Initially, for a given set of shared resources and application objects, the coordination and security policy for a collaboration is specified based on a role-based model.

2. The specification is checked to ensure that it is consistent, i.e., no conflicting security and coordination constraints are specified. Moreover, verification also ensures that the specification comply with the security and coordination requirements.

3. From the specification, various policy modules are derived for different requirements, such as role based security, object level access control, and event notification for coordination.

4. Finally, through these modules, the collaboration environment is constructed by a generic middleware.

We term our approach as *policy-driven* construction as the middleware is only responsible to enforce policies derived from the specification. This approach sharply contrasts with the commonly used approaches for building computer based collaboration systems. In the past, a large number of collaboration systems have been built by extending a single-user application to multiuser shared execution model (Li et al., 1999; Roseman and Greenberg, 1996; Lauwers and Lantz, 1990; H. Gajewska and Redell, 1994). Another commonly used approach is to build an environment by

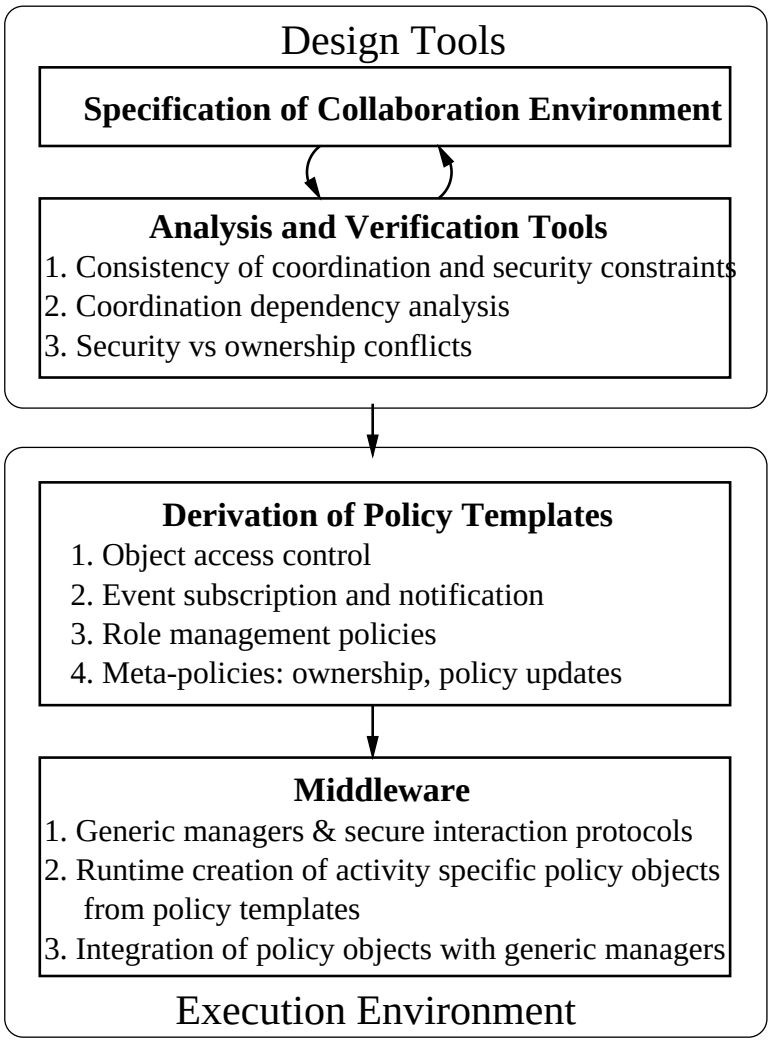

Figure 1.7. Steps in building a distributed collaboration

composition of objects, such as JavaBeans or library modules (Li et al., 1999; Banavar et al., 1998). Usually, such constructions are not driven by any formal model for security and coordination requirements. The objectives of this dissertation is to develop a model for specifying a collaboration environment and describe the security architecture of a middleware to construct the target environment from the specification.

1.6. Comparison with Similar Approaches

In the following sub-sections, the approach of constructing collaboration systems are compared with two programming paradigms: separation of concerns and formal specification-driven development. These programming paradigms, as well as our approach presented in this thesis, can also be called methods for software architecture as these approaches address both design and development of software systems.

1.6.1. Separation of Concerns

A motivation behind our approach is separation of coordination and security aspects and specify them separately. Similar approach is motivated by separation of concerns (Parnas, 1972; Ossher and Tarr, 2001), which is a principle utilized for software development to manage large scale software projects. In the following, several methodologies or programming paradigms for separation of concerns are discussed in comparison with our approach.

Separation of Control

Separation of control is an design concept, which is motivated by open implementation in contrast to traditional black box implementation of software modules (Kiczales, 1996). In black box implementation of modules, only interfaces of the modules are visible to other interacting modules providing abstraction by encapsulating underlying implementation. Open implementation is proposed in contrast so that module implementation can be fine tuned based on application requirements. Separation of control is the central idea in open implementation. The mechanism behind this idea is the concept of computational reflection, which explores meta-interfaces for modules so that the underlying implementation of a module can be changed. The approach presented in this thesis also seeks separation of control. However, our motivation is to adapt with changing security and coordination policies, as oppose to application context specific fine tuning of object implementation for better performance metric.

Separation of Coordination

Linda (Gelernter and Carriero, 1992) is coordination language, which motivates the separation of coordination from computation. Computation can use Linda primitives

to create computational activities and to support communication among them. To change the coordination policies, the use of Linda primitive within computation has to be changed. Linda primitives are strongly integrated with the code for computation. On the other hand, our specification rely on object interfaces to specify coordination policies.

Aspect Oriented Programming

Aspect oriented programming (AOP) concepts have evolved to handle specific design decisions termed *aspects*, which are hard to capture in code using object-oriented methodology (Kiczales et al., 1997, 2001). The main motivation behind aspects is that they cross-cut a system's basic functionality and cannot be encapsulated in a generalized procedure or class definition. Based on system domains, aspects can be failure handling, synchronization constraints, result sharing, or even compile time memory allocation. In AOP, aspects are separated from the rest of the program, which are termed components. In addition to a language to program the components, AOP implementation of an application requires *aspects languages* to program the aspects. An *aspect weaver* combines the component and aspect languages. In AOP, *join-points* are the elements of the component language semantics that the aspect program coordinate with. The aspect language resembles meta-program, which uses *join-points* information, and the aspect weaver functions on the join-points. An example of aspect-oriented tool is AspectJ (Kiczales et al., 2001), which is an aspect-oriented extension to Java. Our approach sharply differs from aspect oriented programming as we are concerned with coordination and security aspects from the perspective of application users rather than application developers. The coordination and security specification can be modified by the collaboration designer based on object interfaces without a development cycle. Compared to aspect languages, our specification is supported by a specification model and is verified against the requirements.

1.6.2. Specification Driven Construction

Formal specification is widely used at design phase for understanding requirements (Kotonya and Sommerville, 1998). Executable specifications (Gravell and Henderson, 1996; Fuchs, 1992) are used for software engineering of systems, mainly

to automate construction of prototype systems, for requirement analysis. Similar approach of specification based prototyping is utilized in systems, such as control systems (Heimdahl and Thompson, 2000). Such prototyping ensures that the formal specification is always consistent with the behavior of the prototype as the specification is updated when the prototype evolves.

In collaboration systems, only few systems, such as COCA (Li and Muntz, 1998) and DCWPL (Corts and Mishra, 1996), have taken the approach that is similar to ours of building a collaboration environment by coupling its high level specifications with a general purpose middleware. There has been an effort underway to develop specification languages for enterprise-wide workflow systems (Workflow Management Coalition, 1999), but these efforts are for integration of cross organizational workflow processes.

A motivation behind our approach is to adapt with the changes in collaboration environments, by separating the coordination and security policies from the implementation of the shared objects and providing a wide selection of policies based on pre-designed collaboration specifications. There are other approaches of adaptation in collaboration systems. In EGRET (Johnson, 1992), a framework with predefined data structures for collaboration is presented where these data structures can be manipulated for adaptations. Similarly, in Oval (Malone et al., 1995), a *tailoring language* is provided to the end users to adapt with cooperative work. In workflow systems, adaptation through exception handling are presented as a solution (Kammer et al., 2000; Ellis and Keddara, 2000).

1.7. Organization of the Thesis

The next chapter presents the security requirements in distributed collaboration systems. Chapter 3 presents existing security models for security policy specifications and their deficiencies. The specification model that is developed as part of this dissertation is presented in Chapter 4. Chapter 5 presents the verification methodology of collaboration specifications using this model. Design challenges for the policy-driven middleware and the architecture of a prototype implementation are discussed in Chapter 6. In Chapter 7, a wide range of sample collaboration systems and their specifications, together with an evaluation of our approach, are presented.

Chapter 2

Security Requirements in Collaboration

In CSCW and workflow systems, traditional security requirements of integrity, availability, and confidentiality are naturally present. However, these security requirements in CSCW and workflow systems have application level as well as organizational level attributes. Greif and Sarin (1987) noted that the protection mechanisms commonly provided by operating systems and database systems tend to be largely inadequate for collaborative applications. Such applications require protection mechanisms to support convenient realization of application-oriented security policies. For example, in a collaborative software engineering application, users assigned to review code may only modify the comments section of the code, whereas a developer can modify any sections of the code. Moreover, the code review can be performed at a code review phase, which can only start after the development phase. These policies widely vary from one organization to another, and they may also change with new software development methodologies. Hence, policy specification and protection mechanisms are usually build within the application. Based on the management aspects – such as small group, intra-enterprise, inter-enterprise – and the deployment environments, i.e., intranet or Internet, the security requirements of the distributed CSCW and workflow systems vary.

Security requirements in multi-user systems can be traced back to Clark and Wilson in 1987 (Clark and Wilson, 1987). They emphasized on two fundamental concepts to ensure integrity of data. These are (1)"well-formed transaction" ensuring that data can only be modified based on predefined constraints, and (2) "separation of duties", which ensures that conflicting users cannot modify data. Later, various form of "separation of duties" and "Chinese Wall Security Policy" (Brewer and Nash, 1989) have been addressed by commercial workflow systems.

Organization level security policies are usually specified based on the role of a specific user group. Roles (Sandhu et al., 2000) represent well defined organizational entities, and different users may be assigned to the same role in the lifetime of an organization. The use of role based security policies in CSCW and workflow systems has been found to be quite natural as participants perform a set of well-defined tasks pertaining to their expertise and responsibilities in the organization. In a role based security model, a role represents a set of privileges. A user assigned to a role acquires those privileges.

In the following sections, various aspects of the security requirements in distributed workflow and CSCW systems are discussed.

2.1. Separation-of-Duties

"Separation of duties" policies are utilized in organizations for integrity of business processes to properly address and circumvent conflict-of-interest situations. "Separation of duties" policies can also mandate that more than one user is involved in different stages of critical business processes. For example, in an invoice processing workflow, a "separation of duties" policy can be that the invoice preparer and invoice approver are two distinct users ensuring a user cannot approve his/her own invoice. There are various forms of "separation of duties" policies (Simon and Zurko, 1997; Nyanchama and Osborn, 1999; Jaeger and Tidswell, 2001; Sandhu, 1988; Tidswell and Jaeger, 2000; Gligor et al., 1998). The widely used variants are presented below:

Role based static separation-of-duties: This requires that two given roles should never be assigned to the same person. For example, a user cannot be both the accountant and the manager for an organization.

Role based dynamic separation-of-duties: The static "separation of duties" may in some organizations turn out to be overly restrictive. In the above example, once a user is assigned to either of the accountant or the manager roles, he/she cannot join the other role even though there can be multiple monetary transactions with different business entities. Hence, the dynamic "separation of duty" requires that two given roles cannot be assigned or activated concurrently by the same person. This enables a participant to be a manager for one transaction and an accountant for another transaction.

Identity based separation-of-duties: Enforcement of "separation of duties" policies may require information, such as users' identities, that are acquired from sources outside the policy enforcement mechanisms. An identity based separation-of-duties can require that two particular users should not be assigned to the same role. For example, both the spouses may not be assigned to the same decision making group. These types of policies may require that a specified user should never be assigned to a given role. For example, a rogue user may not be assigned to a system administrator role.

Object based separation-of-duties: This specifies that a user cannot perform multiple operations on the same object by participating in two different roles. A purchase order may not be prepared and approved by the same manager. An "object based separation of duties" policy on the purchase order ensures that two distinct managers are involved in writing and approving the purchase order.

Operational separation-of-duties: The "operational separation of duties" requires that no single participant of a role can perform all the operations related to a business

transaction. Instead of specifying constraints based on the lifecycle of an object, these types of policies concentrate on tasks within the lifecycle of a business process. For example, an accountant can prepare tax-filing and approve the tax-filing, where the same accountant cannot perform both the tasks of the tax-filing transaction. Compared to "object based separation-of-duties" that specify constraints on the lifecycle of an object, "operational separation of duties" specify constrains on the lifecycle of a business process.

History based separation-of-duties: This imposes predefined order on the actions performed by roles. "Operational separation of duties" is also a form of "history based separation of duties". Other examples include where a user can only perform a task if he/she has performed another set of tasks. For example, a user can only agree to an online-privacy agreement after he/she has read (reached the end of the agreement statement using a web browser) the privacy statement.

2.2. Chinese Wall Security Policy

Chinese Wall security policy is another widely utilized policy to resolve conflict-of-interest situations that arise when data from different organizations are accessed by a user. This type of policy is expressed to ensure that once a user accesses some resources, that user cannot access any resource that would otherwise create a conflict-of-interest situation. Financial institutions, such as bank, enforce this type of policy so that an agent cannot access financial data of conflicting clients.

Chinese Wall security policy puts constraints on a subject's access of data based on the subject's current access rights on other data. The datasets are grouped into "conflict of interest classes" and a subject is allowed access on only one dataset of each of the conflict of interest classes (Brewer and Nash, 1989). For example, datasets for a financial institution can be grouped into two conflict of interest classes: *Oil Company* and *Insurance Company*. The class *Oil Company* has datasets for multiple companies, e.g., A, B, and C and the *Insurance Company* has datasets for companies X, Y,

and *Z*. Once an agent of the financial institution accesses the dataset *A*, the agent is prohibited from accessing any other datasets of the class *Oil Company*; however he/she can access a dataset, either *X*, *Y*, or *Z*, of the class *Insurance Company*.

2.3. Confidentiality and Privacy

Confidentiality and privacy policies are related to information flow constraints. Traditionally, information flow policies in military information systems protect confidential data. In these systems, mandatory access control (MAC) policies ensure that data can only flow on a predefined path of subjects that are classified from low to high clearance levels, such as *Public*, *Military*, and *Secret Service*. Data are also categorized to specify policies on categories of data that can be accessed by specific classes of subjects. Systems for dynamic coalition also require similar confidentiality policies on critical data shared among collaborators in distributed domains.

Confidentiality policies are often distinguished from privacy policies in that confidentiality policies express the interest of organizations where privacy policy protects the interest of individual user (Anderson, 1996). Privacy is also defined as having control over information about oneself (Tavani and Moor, 2001). In many systems, including healthcare information systems, the terms are utilized interchangeably.

In recent CSCW systems, security related to presence-awareness and privacy has become a concern. In contrast to operating systems and database management systems, which tend to hide the presence of one user from another, a collaboration system is required to support user-presence awareness. 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 be only visible through his/her role or a pseudonym but not by name. Pseudonyms may be required when a particular member has performed a role's task and in future he/she may need to be referred to as part of the policy specification. For example, the identity of a reviewer of a conference paper needs to be hidden from the paper authors though the authors are

able to direct questions to particular reviewer.

Similarly, the working or the protocol of a collaborative activity may need to be hidden. For example, if tasks are assigned to its participants in a round-robin manner, this information may need to be hidden so that a task requester is not able to use this information for his/her own advantage. A similar collaboration requirement can specify that only the owner of a role or group knows the identities of the role members. For example, in a conference review process workflow, the *Program Chair* role owns the *Reviewer* roles reviewing the submitted papers. However, none other than the users in the *Program Chair* role are permitted to view the identities of the members of the *Reviewer* roles.

To protect user specific data collected by online service providers, the Platform for Privacy Preferences (P3P) (W3C, 2002a) provides recommendation for the provider to publish agreements on the way the data is collected, used, and stored. Hence P3P privacy policies not necessarily limit information flow initiated by subjects but agree on certain aspects on collection, usage, storage, and sharing or distribution of data related to user interactions with a specific service provider.

The "Standards for Privacy of Individually Identifiable Information" promulgated by the Department of Health and Human Services (Department of Health and Human Services, 2004) regulates how personal information can be shared among various parties in healthcare services. The regulation sets boundaries on the use and release of individual data and holds violator accountable. A primary aspect of security policy in privacy domain is the control of the user on the data that relates to him/her-self, similar to *Originator Controlled Access Control* (Graubart, 1989) policy. These privacy policies are expressed in terms of *consent*, *obligation*, and *data category* and *context*. For example a patient's consent is required when data is released to a different hospital. An obligation of the healthcare provider can be that all the access to the patient data will be logged. An example policy based on data category and context is that only the patient's billing information can be disclosed to insurance providers.

A related challenge in preserving privacy is to sanitize the data, when used for research or statistical reports, so that information cannot be linked with individuals. Research in database security (Lunt et al., 1990) proposed solutions based on hiding related data in relational databases. Due to relational nature of the data and meta information of the data, traces of the hidden data remain. For such cases, replacement with false data is proposed. However, these types of solutions are proposed for centralized database systems. In current environment of distributed service providers, data is shared with organizations in different administrative domains. This introduces additional policy requirements to specify usage restrictions on the shared data. For example, an organization may like to specify a policy that shared data can only be used for research purposes and cannot be shared with other parties.

2.4. Context Sensitive Security Policies

Policies related to security attributes – privacy or confidentiality, integrity, and resource access – in CSCW and workflow systems need to handle various context-sensitive aspects of the collaboration environment. Such context can be physical location, coordination state of the cooperating users, proximity to devices, or any other application defined contextual information. For example, in a healthcare activity, physician can only view certain test results only during the surgical procedure.

Access rights, privileges, and ownership of objects may change in collaborative environments as activities progress. Sometimes permissions may change due to the user's own actions, such as making a final agreement by signing a contract. For example, only after final submission of a tax filing request by a user, an accountant can view the submitted information.

Several types of "separation of duties" constraints and history-based access control conditions also fall into the category of dynamic access control policies. Additional context-based access control may be related to physical environment's events. For

example, in an organization, context based access control may specify that the managers can access employee files only when they are physically present in a specific room and that too only during a predefined period.

2.5. Security Policy in Workflow Systems

A workflow involves well-defined work activities or tasks, and security policy is expressed based on who can perform specific tasks. Security policies that are expressed based on tasks, such as "operational separation of duties" and Chinese Wall policy, are widely utilized in workflow systems. Task-based security policies are also constrained based on time period. For example, a workflow security policy can be that invoices can only be approved on Fridays.

The security policy in workflow systems has many constraints representing "well formed transaction", as expressed by Clark and Wilson (Clark and Wilson, 1987). An example of "well formed transaction" is the constraint that requires that entries in two accounting books are updated for each banking transaction. This provides integrity of data when one of the entry is corrupted. Other policies in workflow systems represent similar constraints. For example, in a banking system, a vault can only be opened with simultaneous presence of two bank managers. In many cases, these policies are derived from organizational level risk management strategy. For example, bank tellers require concurrent approval of another employee to transfer money over the limit of a certain amount.

Recent BPM systems, including E-commerce, B2B, and Web Services, have introduced various security requirements. Confidentiality requirements in an online auction with sealed-bids may require that the bidder identities to be kept secret. Online job-search and other match-making services provide confidentiality ensuring that match-making is performed based on only non-identifiable attributes of the clients (Jajodia et al., 2001). E-commerce that brings together multiple parities – such as buyer, seller, bank, and insurance – need to ensure that only required information can be

accessed by the parties involved. For example, the bank cannot access description of the product and the seller cannot access the bank account. Another form of access control, termed *usage control* (Park and Sandhu, 2004), is also discussed in current workflow systems, so that the subject's access to a resource is revoked after usage limit. For example, an online service provider may only allow usage up to 30 hours.

In E-Services, anonymous service, both the anonymity of service requesters and the anonymity of service providers are important security requirements. For example, a patient may like to get health related medical information without revealing his/her identity. On the other hand, service providers may like to sell items or services without their identity revealed.

For E-Commerce, Secure Electronic Transaction (SET) (Kawatsura, 2003) protocol provided security services, such as authentication, authorization, integrity, non-repudiation, and confidentiality, for multi-party financial transactions. However, such protocols are limited to well-understood security requirements in specific service domain. For multi-domain business orchestration, such as B2B and Web services, interoperability of security policies of different organizations and systems is a major concern. Security aspects of service provisioning need to address agreements among these domains to adopt a standard vocabulary to express security requirements and a common transport mechanism to interact.

To solve these challenges, Web Services has adopted Extensible Markup Language (XML) to resolve all interoperability concerns, from policy specification to message transport. XML provides the facilities to define tags to express any structural content using text. Based on domain knowledge, XML schemas are developed to represent such structures, and these schemas are shared among interacting systems. Several security related schema standards for Web Services security have been developed. Among them, Security Assertions Markup Language (SAML) (OASIS Security Services TC , 2003) provides a schema for Web Services to exchange information related to authentication and authorization using trusted statements, termed *assertions*. In

Workflow	CSCW
Task based security policies	Unobtrusive coordination
Well formed transactions	Short−term credentials
Seperation−of−duties	Fine−grain access control
Chinese Wall security policies	Contex−sensitive access control
Privacy agreement	Privacy in presence awareness
Anonymous service	
Usage control	

Figure 2.1. Attributes of security policies in workflow and CSCW systems

SAML, there are three types of security assertions: (1) authentication assertions issued by authentication servers, (2) attribute assertions, by attribute servers, related to attributes required to access a service, and (3) authorization assertions, which are generated based on the previous two types of assertions, to access a service. A service request in Web Services may require interactions of multiple service providers, and SAML provides assertion, similar to "single sign on", to interact with these services.

2.6. Security Policy in CSCW Systems

In CSCW systems, security is usually ignored to emphasize the motivations of cooperation and shared objectives. However, several research papers and field studies, namely (Kling, 1991; Kyng, 1991; Orlikowski, 1992), pointed the need of control and existence of conflicting goals among participants in groupware systems. Certain collaboration activity, such as a collaborative preparation of settlement documents by lawyers, is inherently adversarial (Cohen et al., 2000). In many existing groupware literature, a coordination constraint is viewed as an access control constraint and a clear separation of coordination and security policies is not present. In shared-view

GUI oriented groupware systems in regard to unobtrusive coordination, such as preventing "scroll-war" (Dewan and Shen, 1998a) is addressed as a protection issue. In Suite (Dewan and Shen, 1998a), a collaborative editor, the scroll bar is a shared resource, and the collaborating users can manipulate it on their discretion. When the collaborative users block each other on their usage of the scroll bar, without following social courtesy, it is termed a "scroll-war". All the participants of a collaboration may not be identifiable, and they may not follow social courtesy of not engaging in a "scroll-war".

Security policy can be argued to be distinct from coordination policy. Access control policy denies access to information and restricts the flow of information. Security policy in a collaboration treats its participants as adversarial. On the other hand, coordination policy assumes the participants of a collaboration are cooperative for their best interest to achieve a goal. The coordination policy exists to facilitate the participants to share resources in an agreed upon manner to reach that goal. Hence, security policy needs to be specified for resources in a shared environment irrespective of the presence of a collaboration.

As in workflow systems, in groupware systems, users also perform their tasks based on well defined roles. For example, in a whiteboard sharing activity, users can be in *Moderator*, *Writer* and *Viewer* roles and acquire corresponding privileges for role specific tasks. However, unlike workflow, a user's roles may represent short-term privileges. For example, *Writer* role may be assigned to different users, one at a time, to coordinate sharing of the whiteboard. Another distinction is that security policies in groupware are often based on the attributes or the structure of the shared resources. This in turn requires fine-grain access control policies (Dewan and Shen, 1998a). In collaborative design, different users may access or modify different parts of a shared object. Group-awareness technologies, such as presence-awareness in Instant Messaging, has introduced security policies regarding who can acquire the presence data, such as "buddy-list". This type of policies are a tradeoff between unwanted interruption and enabling discovery of group activities. In video conferencing tools,

hiding visible objects surrounding the communicating users is also addressed as a security issue to ensure that unwanted information does not leak (Hudson and Smith, 1996).

2.7. Meta-Level Administrative Security Policies

In a decentralized execution environment, participants from domains with mutual distrust need to manage the shared resources, activities, and participants' roles in a collaboration. Policies related to identity management, object access, task creation and coordination, and other administrative aspects of managing collaborative activities require to be enforced in security domains managed by *administrators* who are *trusted.*

There are mainly two distinct forms of trust that are discussed in the context of distributed systems. First, trust relations are defined for distributed authentication of identity related certificates for encryption and access control (Blaze et al., 1996). Second, trust is defined as an entity's "trustworthiness" and discussed in the context of deriving recommendation or reputation (Abdul-Rahman and Hailes, 1998; Azzedin and Maheswaran, 2003). This type of trust is termed as *behavior trust* (Azzedin and Maheswaran, 2003). In addition to these trust concepts, in a decentralized collaboration environment, trust needs to be assigned to entities for administrative task of enforcing policies and performing management functionalities that may arise at runtime.

In most of the existing systems, administrators are trusted not to violate polices under their control. When administrators are in domains with mutual distrust or lack of trust, meta-level policies are required specifying facts regarding trust among these cooperating domains. Meta policies are needed to be specified for administrative roles that are trusted to enforce policies. Examples of such meta policies include specifying who can be present in these administrative roles.

In collaboration environments, such as ad hoc collaboration of mobile users and

peer-to-peer systems, there may not be any dedicated administrative entity. Rather each user represents the administrator of its own unique domain. For example, in a conference review meeting, participants may form a collaboration using their own mobile computing devices and collaboratively maintain review decisions and corresponding reports. Similarly, in peer-to-peer systems, a user may trust other online users to store his/her resources. Trusting strangers to form such collaborations raises security policy questions that are traditionally addressed as part of the security policy enforcement mechanisms. For example, identification of users based on their off-line credentials and accepting resources from users whose identities may not be verified.

In distributed environments, due to the lack of cost effective solutions of cryptographic services, the users are prompted to choose from a wide variety of cryptographic protocols, algorithms, and related attributes, such as, the length of the shared secret key and the strength of the cryptographic protocols. Often these decisions are pushed to the user level as security policy requirements. For example, group session security requirements are addressed by GSAKMP (Harney et al., 2001).

2.8. Related Work: Access Control Models in CSCW

Suite (Dewan and Shen, 1992, 1998a,b) deals with a wide-range of access rights on shared data. In this model, a subject can take a role, and role is used for subject aggregation, similar to group concept to ease access specification using access matrix. Later in Emily (Smith and Rodden, 1993), an access control model for multiuser interface is presented where many ideas are similar to Suite (Dewan and Shen, 1998b). However, it mentions that in (Dewan and Shen, 1998b), semantics of the rights are embedded within Suite model, and they are not extendible. This model also extends the access matrix where access and shared flags are maintained with each object's method. The shared flag indicates if the result of applying a method by one user is to be propagated to others.

MAP (Management Authorization for site Policy) (Zurko and Simon, 1996) emphasizes usability of access control mechanisms. MAP augments an ACL mechanism

with sensitivity labels, object groups, and access rules. It states that existing Military Multi-Level Systems (MLS) are hard to use and prone to "label float"- information gets over-classified as a by-product of the mismatch between the ways users produce or use information and how labeling is determined by the MLS model. This paper states that, in groupware, the correspondence between privileges and tasks are usually not defined by the system resulting in ambiguity by the users.

An approach of assigning discretionary access to visiting mobile agents in agent based collaborative systems is addressed in (Jaeger and Prakash, 1996). The solution is more tied toward access control for mobile agents than CSCW systems. In BSCW (Sikkel, 1997), a role is a user group that requires additional authentication. Certain roles may require explicit role admission, e.g. superuser. To grant such a role, additional conditions, like user must be working from a workstation within the building, may be specified. This paper presents single-step delegation, recursive delegation, and delegation within a trusted group.

Team based access control (TMAC) (Thomas, 1997) is an approach of applying RBAC in collaborative environments such as those involving workflow. In addition to role-based permission assignment for users and object types, TMAC proposes activation of permissions for individual users and object instances. This is an active security model, which proposes a context sensitive permission activation constraints which are lacking by generic RBAC but supported by trigger oriented active databases. A team has participants from different roles and users, and teams can be dynamically activated and deactivated.

SPACE (Bullock and Benford, 1999) presents a navigational or location dependent access requirements for groups. It specifies requirements like presence of multiple users, maximum or minimum number of users, certain number of users from certain

groups, and combination of these to provide access on certain 3D and 2D collaborative spaces.

2.9. Summary

In this chapter, we have discussed security requirements related to availability, integrity, and confidentiality in CSCW and workflow systems. Security requirements that are addressed in contemporary CSCW and workflow systems are compared. Primary differences between traditional group-oriented system and contemporary collaboration systems are identified in this chapter.

Chapter 3

Security Models

Security policy is the specification of security requirements, usually specified based on some security model (see Figure 3.1). A security model usually represents a particular set of policies (Bishop, 2000). Traditional security models are classified in two groups: Discretionary Access Control (DAC) and Mandatory Access Control (MAC) models. In DAC models, access right can be changed on discretion of a user, for example access control model in a UNIX file system. On the other hand, MAC models represent mandatory policies where access control are enforced by trusted organization level enforcement mechanisms, and an individual user does not have control over changing access rights. For example, lattice based military access control models.

As security models are developed based on security requirements of particular system, often these models are specific to security properties, e.g., Clark and Wilson model for integrity, Bell and LaPadula model (Bell and La Padula, 1975) and its derivative lattice based models for confidentiality, and take-grant model (Snyder, 1981) for access leakage. Though these models have various formal results regarding the types of security properties they support, there are several concerns:

1. The security model needs to be expressive to specify a wide range of security policies, such as separation-of-duties and other context dependent security policies.

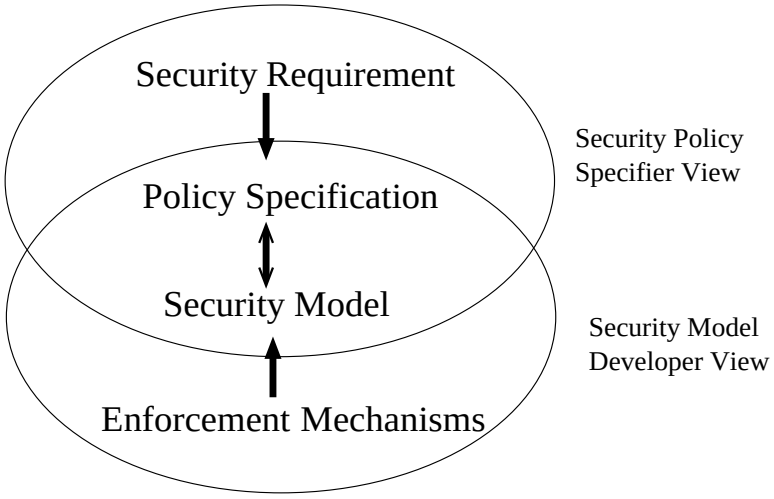

Figure 3.1. Security requirement, policy, model, and enforcement mechanisms

2. Administrative capabilities to manage security policies is a major usage concern. Management of security policies is a challenge where user population is large and dynamic, i.e., users are joining and leaving an organization. On the other hand, MAC based policies are hard to manage due to the administrative requirements of proper categorization of data and users.

3. Ideally a security model should be able to ensure that the specified constraints do not violate any desired security properties, such as:

   (a) Confidential information cannot flow to unauthorized users;

   (b) Authorized information can be accessed;

   (c) Any temporal or conditional constraints on accessing objects can be satisfied;

   (d) No rights can be leaked to unauthorized users.

   Most of the existing security models, specifically for distributed CSCW and workflow systems, cannot support all of the above security properties.

4. As security requirements may depend on complex conditions, conflicting or inconsistent security requirements may be specified. For example, a user can be member of role `A` if he/she is a member of role B; and a user can be member of role `B` if he/she is a member of role A. This represent a policy conflict as the user cannot be a member of either of these roles. Similarly, an inconsistent policy may specify that a user can access an object if he/she has accessed it earlier. Policy conflict also arise due to granting negative and positive access-rights to the same user. For distributed CSCW and workflow systems, a security policy specification methodology is required that either provides conflict resolution rules (Jajodia et al., 1997) or supports verification of security policies to ensure that the policy specification satisfies security requirements (Ahmed and Tripathi, 2003a).

3.1. Security Model for Access Leakage

Ideally, a security model should be able to ensure that no rights can be leaked to unauthorized users. This property is known as *safety* property. The safety property of the HRU access matrix model (Harrison et al., 1976) that represents generic protection systems is not decidable. On the other hand, safety in take-grant model (Snyder, 1981), a graph based security model, is not only decidable but also has linear complexity that is proportional with the initial size of the access graph. This is due to the fact that take-grant model is restricted to expressing capability based systems, i.e., it only expresses properties related to delegation of rights. Based on this fact, later access control models have placed constraints on access control structures to facilitate analysis of safety properties, such as, Typed Access Matrix (Sandhu, 1992). In other cases, users with administrative rights, e.g., *create-subject*, are trusted for not violating security properties.

3.2. Security Models for Confidentiality

Initially, access control model was proposed as security model for confidentiality. Early example of access control model for confidentiality is the Bell and LaPadula model that imposes mandatory access control on the data that a subject can read or write based on classification of data and subjects. Based on imposed access control restrictions, information can only flow from users with low classification to users with high classification. Access control models are easy to implement with tamper-proof execution monitor, also known as "reference monitor", that interposes on subject's request to access any object (McLean, 1994). On the other hand, this type of model can only enforce confidentiality policies on information that can be leaked through *storage channel*, e.g., disk file or computer memory. Another threat of information leakage is *covert channel* (Lampson, 1973) that is not intended for information transfer but represents effect of the runtime program. For example, the size of the UNIX *tmp* directory can be used to pass information between two confined programs. McLean (McLean, 1994) argued that covert channels are real threats as the capacity of such channels can be large due to the rise of computers' processing speed. Even when the users are trusted not to violate confidentiality policies, *Trojan Horses* can utilize these channels. Trojan Horses are programs that disguise themselves as legal programs but have covert behaviors.

To analyze covert channels as part of security models, *interface model* for confidentiality has been introduced. Interface models specify restrictions on systems input and output interfaces to ensure enforcement of confidentiality properties. Based on the behaviors of systems and covert channels, various types of confidentiality properties for interface models have been introduced, such as noninterference, noninference, and non-deducible (Zakinthinos and Lee, 1997). For runtime enforcement of confidentiality properties, it is argued that an execution monitor to ensure secrecy cannot be complete (Volpano, 1999), due to covert channels. Denial of an execution step itself represents a covert channel. Therefore, static analysis of a system for confidentiality properties is preferred to ensure secrecy.

Model oriented formal methods, such as CSP (Communicating Sequential Processes) (Schneider, 1996), SPA (Security Process Algebra) (Focardi and Gorrieri, 1997), TLA (Temporal Logic of Actions) (Fine, 1996) and state based techniques (Bevier and Young, 1994) are used to statically verify various types of confidentiality properties. In the property-oriented formal methods, theorem provers (Andrews and Reitman, 1980; Martinelli, 1998) and type systems (Denning and Denning, 1977; Millen, 1981; Volpano and Smith, 2000; Simonet, 2002) are used for analyzing information flow. However, these approaches assume that the verified programs will run under trusted subjects. In decentralized administration of systems, programs are type-checked based on assigned "trust" (Ørbæk and Palsberg, 1997) or labeling data (Myers and Liskov, 2000); however, the execution environment is assumed to be entirely trusted. In this thesis, we have utilized finite state based model checking to verify security properties in collaboration systems (Ahmed and Tripathi, 2003b). In our verification approach, some of the distributed security domains can be under the control of administrators who may not be trusted, and the behavior of untrusted administrators are modeled to verify security properties (Ahmed and Tripathi, 2003a).

3.3. Role Based Access Control

Role based access control (RBAC) is policy neutral (Sandhu, 1998), i.e., it is not tied to any specific policy model, such as MAC or DAC, and is able to express security policies related to MAC and DAC models. As the definition of a role within an organization changes less frequently compared with the turn-around rates of people, role based systems are said to provide ease of user management. The primary motivation behind role based access control models is their ability to express various forms of security constraints, such as "separation of duties" and role cardinality constraints. A role cardinality constraint ensures the presence of a maximum or a minimum number of users before a role privilege can be executed.

The concept of roles and related theories have been widely studied in the past

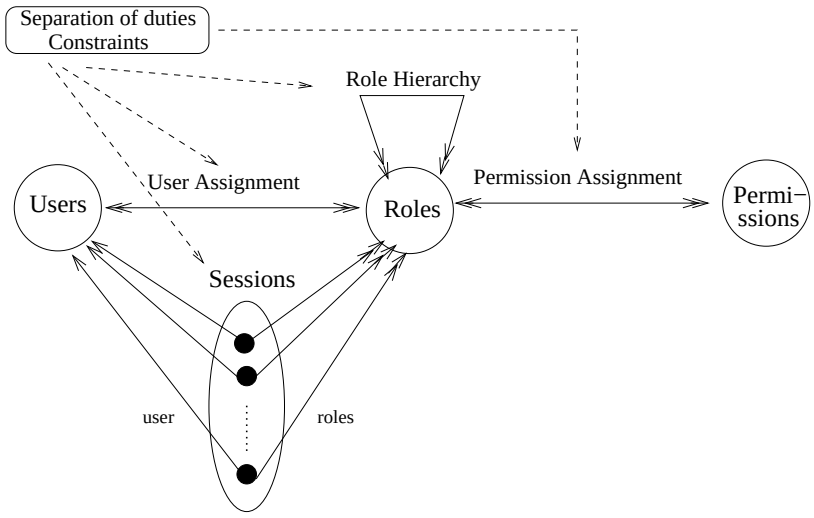

Figure 3.2. NIST RBAC models (Sandhu et al., 2000)

in the context of behavioral science (Biddle, 1966). Due to different interpretation of roles, based on application domains, such as distributed systems (Bacon et al., 2002), network management (Lupu and Sloman, 1997), database management systems (Baldwin, 1990), and interactive groupware (Dewan and Shen, 1998a), a wide range of role based models have been proposed. NIST has proposed a unified standard for role based access control (RBAC) reference models (Sandhu et al., 2000). NIST RBAC models have three primary constructs: *User*, *Role*, and *Permission*, as shown in Figure 3.2. Roles are assigned a set of permissions and users are assigned to roles. A user can have multiple roles, and the same permission can be assigned to multiple roles. The one-to-many relation is shown with a double arrow, where as the one-to-one relation is presented with a single arrow, in the Figure 3.2.

NIST RBAC models support the concept of role hierarchy to define roles where a *senior* role in a hierarchy can acquire all the privileges of *junior* roles. In the Figure 3.3, a *System Administrator* role has the *execute* permission, a *Developer* role has the *write* permission, and a junior role *Reader* has the *read* permission. Both the *System Administrator* and *Developer* roles acquire the *read* permission. In this example

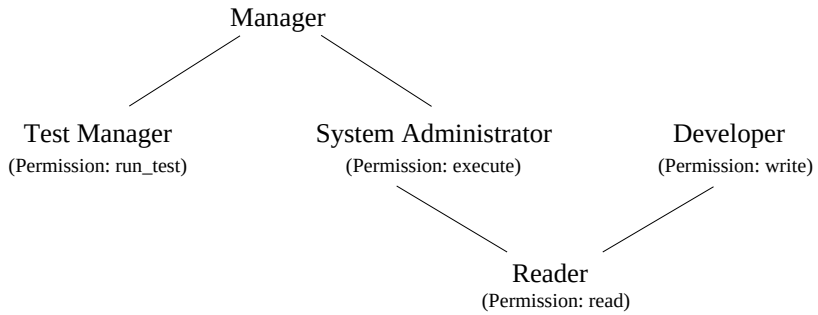

Figure 3.3. Example role hierarchy

role hierarchy, the *Reader* role is declared to contain the common permission *read* for the two senior roles. On the other hand, the *Manager* role aggregates all permissions from the subordinate *Test Manager* and *System Administrator* roles.

To perform some task, a user may require a set of permissions. These permissions may be acquired based on different sets of roles that are assigned to the user. Assume in our example, a user is assigned to both the *System Administrator* and the *Developer* roles. When the user wants to read a file, there are multiple choices of roles that can perform the task. These choices are *System Administrator*, *Developer*, *Reader*, or any combinations of these roles. In RBAC (Sandhu et al., 2000), a user's acquisition of a set of roles to perform some task is defined as a *session*. To decide on the set of roles for a session, different rules can be used. Based on the least privilege principle, a set of roles can be selected that result in least number of permissions. In this example, the *Reader* role has the least number of permissions for the session to read a file.

In NIST RBAC models, "separation of duties" constraints can be specified on the relations of *user assignments to roles*, *permission assignments to roles*, role hierarchies, and sessions. For example, a "static separation of duties" can be realized by a constraint on user assignments to roles.

3.4. Challenges of RBAC Model for CSCW Systems

NIST acknowledges that RBAC is an open-ended technology and does not address all the issues in role based security (Sandhu et al., 2000). In the following subsections, several such issues that are essential in distributed CSCW and workflow systems are discussed.

3.4.1. Role Membership Management

NIST RBAC models do not incorporate administrative issues in user assignments and permission assignments to roles due to lack of consensus. Administrative RBAC that is proposed with *administrative roles* can manage user assignment in a RBAC model (Sandhu et al., 1999). ARBAC declares separate role hierarchies with administrative roles that have administrative privileges of managing users on the roles declared within an organization.

On the other hand, in distributed CSCW and workflow systems, role based model needs to address constraints on users' acquiring a role. Role admission constraints specify the conditions that need to be satisfied when a user requests to join a role. The admission constraints can be based on several different criteria: a list of users that should be allowed to join a role; a list of those who should never be admitted; role membership cardinality specifying the maximum number of users that can join the role; events that must happen before a user can be admitted in a role, such as a predefined time period, tasks performed by others, or previous qualifications requiring that the requesting user has been or is currently admitted in some other given roles. The constraints based on previous qualifications may require that the membership in a prerequisite role is also *valid*. If a role membership in a prerequisite role depends on the user's membership in other roles, then those memberships must also be current. Traditionally RBAC models are centralized and only address user assignment to roles. In distributed systems, these role admission related constraints support the functionality of acquiring and revoking role memberships (Bacon et al., 2002).

3.4.2. Dynamic and Context Sensitive Security Policies

NIST RBAC, and most of the existing MAC and DAC style security policies are static or passive, i.e., they do not depend on time or other events. Moreover, they do not differentiate permission assignment and permission activations (Thomas, 1997). In collaboration environments, different permissions may need to be assigned to roles based on various contexts and events. Moreover, NIST RBAC models do not address any history-dependent constraints.

Active database research has addressed trigger based authorization changes and used the notion of condition-based access control (Mohammed and Dilts, 1994). In recent years, different RBAC models are proposed to address issues related to context sensitive access control constraints. Team Based Access Control (TMAC) (Thomas, 1997) discusses *team* as a context for roles. A *Physician* role can be member of multiple teams, e.g., *Surgery* and *Emergency Room*. Based on the context of the team, *Surgery* or *Emergency Room*, the *Physician* role acquire different sets of privileges. Task based authorization models (Huang and Atluri, 1999; Bertino and Ferrari, 1999) express constraints on tasks that can be performed by a role based on its task execution history.

3.4.3. Intra- Role Constraints

In a role based model, coordination among participants in different roles is referred to as *inter-role coordination*. The primary motivation of this requirement is to enforce precedence constraints among different roles' operations. For example, an *inter-role coordination* requirement in an office workflow can be that the *Accountant* role can pay for a purchase order only after the *Purchase Officer* role has authorized the order.

When multiple users are allowed to be present simultaneously in a role, their actions may need to be constrained based on actions of other participants within the role, which is termed as *intra-role constraints* (Tripathi et al., 2003). Multiple users present in a role can participate either *independently* or *cooperatively*. In *independent participation*, all role specific task-responsibilities are assumed individually by a role

member, irrespective of the presence of the other members, e.g., every member of a conference *Reviewer* role has to independently write a review. On the other hand, when the members in a role are assuming task responsibilities *cooperatively*, their actions need to be coordinated. For example, in a hospital patient ward, several nurses may be present in the role of *nurse-on-duty*. However, some medical procedure on a patient may need to be performed only once by any of the nurses. Another type of cooperation may require a task to be performed by all the members of a role, like jointly opening a bank vault. Moreover, in some collaboration environments there may be no coordination among participants' actions, e.g. in an unrestricted whiteboard sharing.

3.4.4. Policy Specification Methodology: Role Engineering and Constraint Specification

NIST RBAC models do not address role modeling or consistent specification of role constraints. Modeling or engineering of roles is a challenging concern in role based security models. Role has different interpretation based on application domains. In distributed systems (Bacon et al., 2002), role is viewed as a certified capability for authorizing access to services. In network management (Lupu and Sloman, 1997), role based management in proposed with support for *obligation* policy. Obligation policy specifies the actions that a role has to perform when certain conditions become true. In database management systems (Baldwin, 1990), role is termed "Named Protection Domain" and resembles capabilities. Only a few recent research efforts address software engineering methodology to model roles (Epstein and Sandhu, 2001).

In workflow management systems, *task* is the primary construct of modeling workflow processes, and many security constraints are specified based on subjects' authorization to perform specific tasks. In workflow systems (Bertino et al., 1997), constraints are specified with a mapping between roles and tasks. In this model, roles need to be related according to a global order so that roles can be prioritized during task assignment. SecureFlow (Huang and Atluri, 1999) imposes workflow authorization constraints on tasks using *Authorization Template*. An *Authorization Template* 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 RBAC, safety of various role based constraints, such as "separation of duties", have been analyzed with logical expression using rule-based systems (Bertino and Ferrari, 1999; Ahn and Sandhu, 2000; Huang and Atluri, 1999) and graphical models (Jaeger and Tidswell, 2001; Nyanchama and Osborn, 1999; Osborn, 2002; Koch et al., 2002). However, the verification in most of these existing research (Bertino and Ferrari, 1999; Ahn and Sandhu, 2000; Huang and Atluri, 1999) is either performed in the context of centralized management of systems or the participants in administrative roles are trusted to enforce policies without taking into account the mutual distrust in collaborating domains.

3.5. Related Work: Role Based Models in CSCW

In CSCW systems, roles are used for both access control and coordination models. Sometimes it is difficult to draw a clear separation between access control and coordination models. However, we discuss these systems below based on their emphasis.

The use of role-based security in groupware systems was first discussed in XCP (Sluizer and Cashman, 1985), an experimental tool for managing cooperative activity. It represents and maintains relations among persons, roles, actors, and objects with a small language for knowledge representation. This experimental tool asks the users which role the user wants to assume. Later, MPCAL (Greif and Sarin, 1987) expanded the idea for role based access control in groupware. MPCAL uses role as a set of rules determining access to each type of operation on each type of MPCAL objects. MPCAL does not address how different roles interact, but mentions such requirements. For example, after a user in a role performs some action on an object, some other role can perform further actions on that object. Also, it addresses that sophisticated concurrency control mechanisms need to be supported by roles.

Quilt (Leland et al., 1988), a collaborative document production system, protects documents based on the social roles users play in the document production. The collaboration has "style", which determines the actions permitted on various document objects by the recognized roles. New styles can be created and role hierarchy on permission is based on set/subset relation. Similarly, in ICICLE (Brothers et al., 1990), a collaborative code inspection environment, participants are assigned to one of several predefined roles. PREP (Christine M. Neuwirth and Morris, 1990), a co-authoring and commenting editor, states that predefined combination of role and collaboration type will not work. For example, an author role may have multiple members, however, only one member may be permitted to write. This paper also states that premature definition of role could lead to undesirable consequences. Example, at the outset of a project it is not clear who is going to make a "significant contribution" to be an author.

ConversationBuilder (Kaplan et al., 1992) talks about the situated nature of work activities, i.e, activities evolve. It presents an obligation imposing facility, i.e, requesting another user to perform some task, to interconnect activities. SASSE (Synchronous Asynchronous Structured Shared Editor) talks about transition between activities (e.g. brainstorming, research, plan, write, edit, review etc.), requirements of document control methods (e.g. centralized, relay, independent, shared), writing strategies (e.g. single writer, scribe, separate writers, joint writing), and a fixed number of explicit roles (e.g. writer, consultant, editor, reviewer). However, the support for explicit role is left to be handled by social protocols and is not supported by the system.

URS (User-Role Based Security) in a collaborative software development environment is presented in (Demurjian et al., 1993). It groups users in three levels, user classes, types, and roles for ease of aggregated security policy specification. It lacks specification of tasks associated with user roles and policies for their coordination.

Among recent publications, Kansas (Smith et al., 1998), a shared space environment, also recognizes that roles appear naturally in collaboration and shows how dynamic addition of access to roles is needed. It addresses the interrelation between awareness and roles – others may want to be aware of a role and a role may want to be aware of certain things. CoWeb (Collaborative Website) (Guzdial et al., 2000), defines roles as specific concerns and activities associated with individuals who choose to use the CoWeb. It views roles as natural, stable, and long lived. CoWeb tries to recognize role, and refine and support the roles it has recognized. It supports roles like central users, peripheral users, which do not appear until later in the software development process and its usage.

3.6. Summary

In this chapter, we have discussed issues related to security models. The challenges of development of security models and various results in existing security models related to access leakage and confidentiality are discussed. Role based security model is introduced based on NIST RBAC models and the open challenges in role based security models for distributed collaboration systems are discussed.

Chapter 4

A Policy Specification Model

In this chapter, the model that we have developed for specification of coordination and security policies is presented. The development of the specification model is motivated by several types of coordination and security requirements. These are inter- and intra- role coordination, hierarchical structuring of activities, role membership and admission related constraints, "separation of duties", dynamic access control policies, confidentiality, and meta-level administrative security policies for decentralized management of collaboration systems.

The goal of this chapter is to discuss the main concepts of our model – activity and role – as well as present the syntax of the specification model including activity template, role and related constraints, role operation, condition specification using an event model, naming, and scope rules. The concepts of security domain, protection domain, and trust domain are also defined for decentralized administration of collaboration environments. In this chapter, we show how different types of security policies are specified in our model.

Activity and role are discussed in the Sections 4.1 and 4.2, respectively. In Section 4.4, the syntax of the specification model is presented using example specifications. Throughout this chapter, we utilize an example *Course* collaboration, introduced in Section 4.3, to present our specification model. A meta policy specification

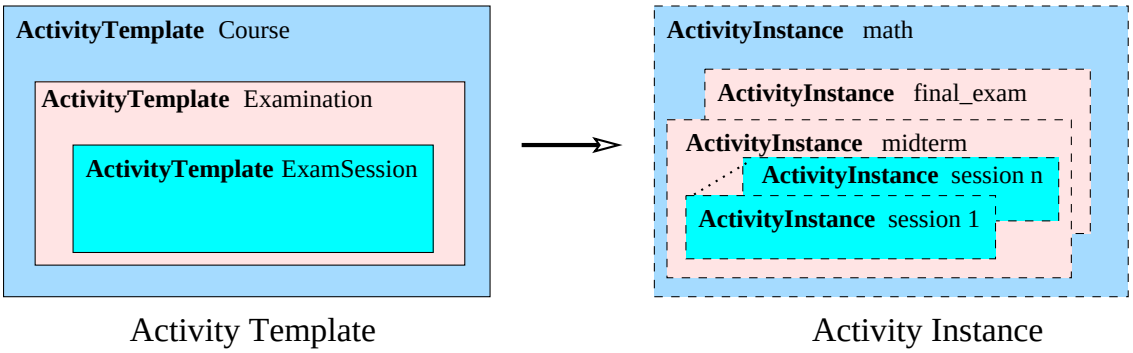

Figure 4.1. An activity template and an activity instance of the template

model for administrative requirements in collaboration environments is presented in Section 4.5. In Section 4.6, the concepts of security domain, protection domain, and trust domain are defined.

4.1. Activity Template

In our collaboration model, an activity defines how a group of users cooperates toward some common objectives by conducting their individual tasks on a set of shared objects. In an activity, users are represented by their roles, and roles are assigned privileges to perform certain tasks.

An *activity* is an abstraction of a collaboration session, which provides a scope for objects, roles, and privileges in the collaboration. In an organization, many activities exist and new activities are constantly created. An activity can also span across multiple organizations. Existing roles and new roles participate in these activities. A large collaborative environment may sometimes need to be structured hierarchically. A collaboration specification needs to support decomposition of tasks into activities/sub-activities and planning how such activities can be performed. Activity decomposition and planning are widely used in collaboration systems, specifically in workflow environments (Hollingsworth, 2004).

An *activity template* specifies a generic collaboration pattern for some specific kind of collaborative tasks involving a set of roles using some shared objects. Any number of instances of a template can be dynamically and concurrently created. An activity template is a collaboration specification. An activity template can be structured hierarchically, consisting of multiple nested activity templates. In Figure D.1, on the left, an activity template *Course*, with nested templates *Examination* and *ExamSession*, is presented. On the right of Figure D.1, an activity instance, named *math*, of the template *Course* is presented. The *math* activity has nested *Examination* activity instances: *midterm* and *final_exam*. In an examination activity, there can be many exam-session activities.

In the nested structuring of activity instances, an inner activity instance is called a *child* activity, and an outer activity instance is called a *parent* activity. In Figure D.1, for the *midterm* activity, the inner *session 1* activity instance is a child, and the outer *math* activity instance is the parent. There can be many concurrent child activities based on the same nested activity template; however, there can be only one parent activity instance for a child.

4.2. Role

In our model, a role defines a set of *operations*. As part of an operation *permissions* are granted to perform the role specific tasks A role operation has a precondition that must be satisfied when an operation is invoked.

Roles are defined in the context of an activity. Objects can be passed into nested activities, and users in roles from the parent activity can be statically or dynamically assigned to roles in nested activities.

Associated with each role, there can be three types of role constraints:

- *Role admission constraints* specify the conditions that must be satisfied when

a user is admitted to a role.

- *Role validation constraints* must be satisfied not only during a user's admission but during the user's whole lifecycle in the role.When these constraints are not satisfied for a user, the user's membership to the role is revoked.

- *Role activation constraints* must be satisfied for performing any role operations.

These constraints and preconditions facilitate context-sensitive access control policies, as discussed later.

4.3. Example Specification: Course

In this section, we present an example collaboration environment. throughout this chapter, using this example the specification model is presented.

Figure 4.2 presents the *Course* activity template with the roles and shared objects. The course activity contains three roles: *Instructor*, *Assistant*, and *Student*. These roles have disjoint members. In a *Course* activity, after the *Instructor* initiates a *BulletinBoard*, all the roles within this activity can share it. A course activity also has a nested *Examination* activity with four roles: *Grader*, *Examiner*, *Examinee*, and *Approver*. The *Instructor* starts an *Examination* activity by assigning members to the *Examiner* role. Within an *Examination* activity, *Examiner* set the exam-paper and the *Approver* approves it. Each examinee creates an instance of the *ExamSession* activity template to take the exam. An exam-session activity contains the roles *Candidate* and *Checker*. An *ExamSession* activity terminates when the *Checker* performs the *Grade* operation. An *Examination* activity terminates when exam-sessions of all the examinees terminate.

Figure 4.3 presents the task flow requirements in the *Course* activity. In the figure, the roles that perform the tasks are not shown. Only the task names and task-flow dependency among the tasks are shown. These tasks are represented as operations

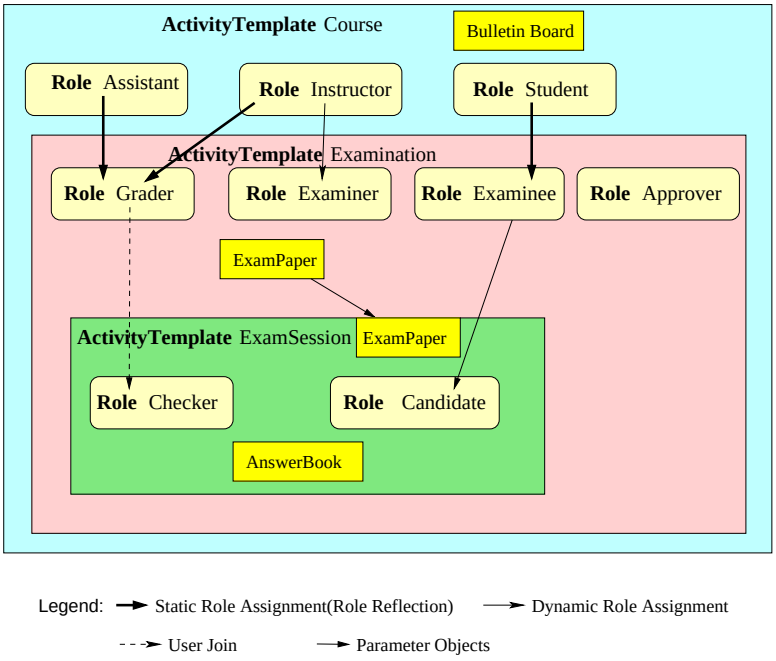

Figure 4.2. Role member assignment and object passing in hierarchical structuring of activities

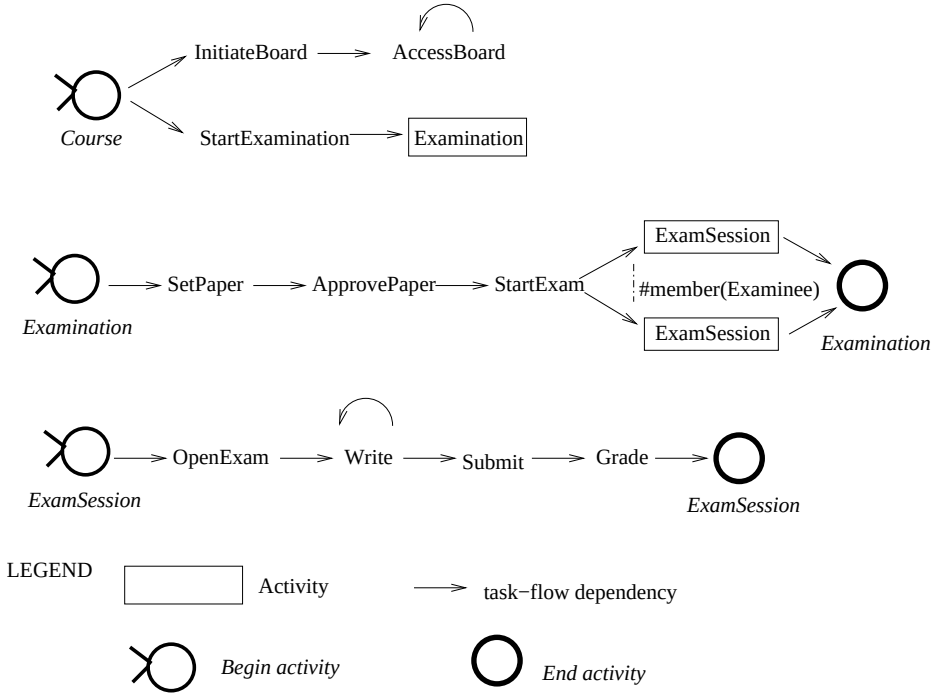

Figure 4.3. Task flow requirements in a Course activity

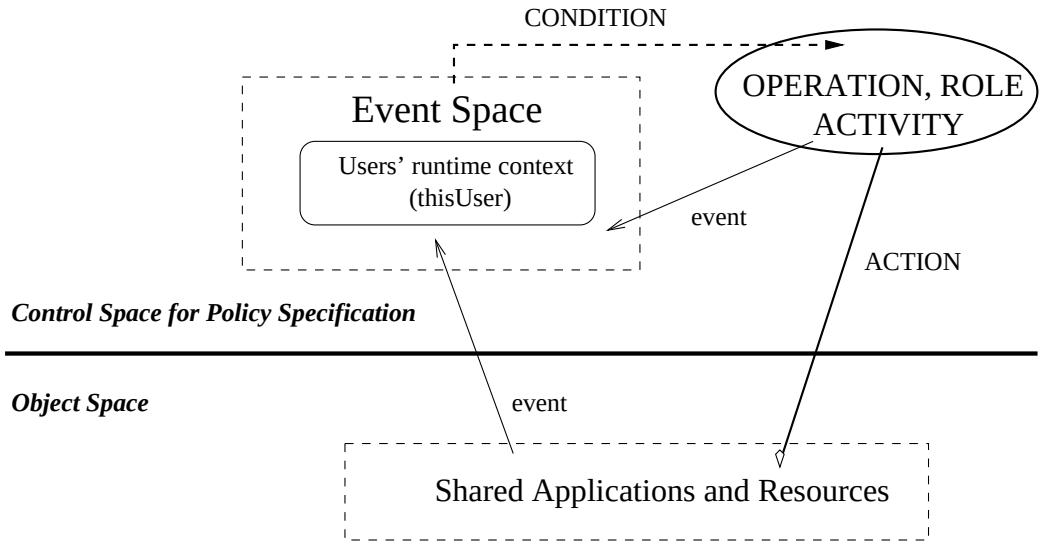

Figure 4.4. Relations among entities in the specification model

and activities in our specification model.

4.4. Specification Model

Figure 4.4 presents the constructs required for the entities, such as activity, role, operation, action, object, and event, in the specification model. As part of an operation's action, permissions on the shared objects are granted, objects and activities are created, methods on objects are invoked, and role members are assigned. The system implicitly generates events for operation execution and activity instantiation. Collection of these events creates an *event history*. The event history is utilized to specify various constraints on role membership management and operation precondition. In Figure 4.4, a solid-line separates the "object space" with shared applications and resources from the "control space for policy specification". The control space contains the constructs for the specification model, i.e., activity, role, operation, condition, and action. As attributes of an event, an operation can import an object's state in the event space. The event space contains a user's runtime context, i.e., the identity

of the user who is currently executing an operation. The purpose of the event-space is to express conditions for coordination and security policies.

In the following subsections, specification model for activity template, role, and operation as well as the model for condition specification using operation events and role membership based predicates is presented. Later sections present how these conditions are used to express various forms of constraints for collaboration systems.

4.4.1. Activity Template Specification

Based on our collaboration model, we have devised an XML schema for collaboration specifications. The XML schema can be found in Appendix B. However, rather than using XML, we use a notation that is simple to read and conceptually easy to follow. In Figure 4.5, the syntax for the XML schema for *activity template* definition is shown, where `[ ]` represents optional terms, { } represents zero or more terms, | represents choice, and boldface **terms** represent tags in XML schema. In this section, we present the collaboration model using parts of the schema. The complete schema can be found in Appendix A.

Based on the definition in Figure 4.5, an activity template must have an id and at least one role defined. Moreover, the template can have other nested templates, a termination condition, passed objects' ids and Java classes that uniquely identify the types of the passed objects. Moreover, an activity template specification can have *assigned roles* and an *owner*, which are explained in later sections.

In Figure 4.6, a partial specification containing activity and role definition of the *Course* activity template of Figure 4.2 is presented. In this chapter, we incrementally present the specification for the *Course* activity template. Complete specification for the *Course* activity template can be found in Appendix D

ActivityTemplateDef $\longrightarrow$ **ActivityTemplate** templateId [**Owner** roleId]
{ **Object** codebase objId} {**AssignedRoles** roleId}
[**TerminationCondition** *Condition*]
RoleDef {*RoleDef*} {*ActivityTemplateDef*}

Figure 4.5. Syntax for activity template definition

```
ActivityTemplate Course (AssignedRoles Assistant, Instructor, Student) {
  Role Assistant{....}
  Role Instructor {....}
  Role Student {....}

  ActivityTemplate Examination (AssignedRoles Examiner) {
    Role Examiner { ....  }
    Role Approver { ....  }
    Role Examinee (Reflect parentActivity.Student) { ....  }
    Role Grader (Reflect parentActivity.Assistant,parentActivity.Instructor){
    ....  }

    ActivityTemplate ExamSession( ExamPaper exam, AssignedRoles Candidate) {
      Role Candidate { ....  }
      Role Checker { ....  }
    }
  }
}
```

Figure 4.6. Skeleton specification of Course activity template using pseudo code

4.4.2. Naming

In the specification model, a role is defined in the scope of an activity, and an operation is defined in the scope of a role. In the specification model, any entity – such as activity, object, role, or operation – encapsulated in the scope of an activity can be referenced by a fully qualified name. For example, in Figure 4.6, the *Instructor* role in the *Course* activity template is referenced as `Course.Instructor`. Similarly, the *InitiateBoard* operation is defined in the scope of the *Instructor* role and is referenced as `Course.Instructor.InitiateBoard`.

Within an activity, one can refer to its current instance using the pseudo variable *thisActivity* and its parent activity instance using *parentActivity*. For example, in Figure 4.6, the *Grader* role refers to the *Assistant* role of its parent instance using `parentActivity.Assistant`. If not qualified by *thisActivity* or *parentActivity*, the default qualifier *thisActivity* is assigned to a name to refer to the current activity instance for the name. The user executing an operation is identified by *thisUser*. Within a role's operation, one can refer to the role by *thisRole*.

In the specification, a name can be assigned for newly created activities or objects. In the following Example 2, the name `board` is assigned for the objects of type *BulletinBoard* created using the construct *new*.

```
board=new Object(BulletinBoard);
```

4.4.3. Scope Rules and Parameter Passing

Activity is the primary scope that represents a protection domain in our collaboration model. There are several scope rules that restrict access to resources, role membership information, and events generated as role operation. These scope rules restrict the visibility of names declared in a specification and confine access to resources and flow of information. The rules are listed below based on the entities they are imposed on.

- **Object access scope:** Roles are restricted with their access rights in the scope of an activity and can access only objects passed or created within this activity. Objects can be passed to a child activity when the activity instance is created. Objects that are declared as a parameter in the child activity's template definition, as shown with *NewActivityDef* in Figure 4.10, can only be passed to a child activity.

- **Operation event scope:** The operation events are confined within the activity scope. Operation precondition can only coordinate with other operations defined within the same activity instance. This includes the *start* and *finish* events of any child activities created within this activity.

- **Role membership scope:** Role membership related events are not confined within an activity boundary. A role membership constraint can depend on other roles that are defined within the same or any ancestor activity scopes. This allows role membership information from ancestor activities to flow to child activity instances, however, excludes membership information to flow out from children to a parent.

The above rules are enforced by restricting visibility of names. Visibility of operation names, newly created activity and object names, and passed object names are confined within an activity instance. Role names within any child activities are not visible in parent and ancestor activities.

In Figure 4.2, based on the object access scope rule, when an instance of the *Course* activity template is created, members of roles *Instructor*, *Assistant*, and *Student* can access the *BulletinBoard*. However, the *BulletinBoard* is not visible to any nested child activity instances. In an *Examination* activity, a reference to the *ExamPaper* object is passed as a parameter to a nested *ExamSession* activity. A single *ExamPaper* object is shared by all the exam-sessions. On the other hand, a new *AnswerBook* object is created in each exam-session. An *AnswerBook* object created within an exam-session activity cannot be accessed by other exam-session activities.

Condition $\longrightarrow$ *RoleCondition*
| *OperationCondition* | *TimeCondition*
| *Condition BinOp Condition*
| !*Condition*
RoleCondition $\longrightarrow$ *#RoleMemberList Relation Count*
| member(*userId, roleId*)
RoleMemberList $\longrightarrow$ members(roleId)
| *RoleMemberList SetOp RoleMemberList*
SetOp $\longrightarrow \cap \mid \cup \mid \setminus$
TimeCondition $\longrightarrow$ date *Relation String*

Figure 4.7. Syntax for role membership based condition definition

4.4.4. Role-Membership and Event-Predicate Based Condition Specification

As shown in Figure 4.7, there are mainly two types of conditions in the specification model: conditions based on role membership functions and conditions based on event predicates. We also support a function *date* that returns current time of the system for time-based condition specification.

In the specification, a boolean function *member(userId, roleId)* checks if the specified user is present in the given role, and *members(roleId)* returns the role member list. Set operations can be performed on role member lists. A *count* operator, `#`, can be applied on a member list. The count of the members in a role is *#(members(roleId))*.

Events and event counters are used in the specification model for specifying coordination and dynamic security policies. Conditions for activity termination, role admission constraints and operation preconditions are specified using event and event-count based predicates.

There are two classes of events that are generated in our model: *system defined events* and *application defined events*. System defend events are implicitly generated

---

OperationCondition $\longrightarrow$ *EventCount Relation Count*
| *EventName*(*Index*)[*AttributeName Relation AttributeValue*]
EventCount $\longrightarrow$ #*eventName* [*AttributeName Relation AttributeValue*]
| *EventCount IntegerOp EventCount*
| *EventCount IntegerOp Count*
EventName $\longrightarrow$ opId.start | opId.finish | activityId.start | activityId.finish
| appDefinedEventId
BinOp $\longrightarrow \wedge \mid \vee$
Relation $\longrightarrow > \mid < \mid = \mid <= \mid >= \mid \neq$
IntegerOp $\longrightarrow +$ | - | `mod` | `div` | *
Count $\longrightarrow$ 0 | 1 | ... | N
Index $\longrightarrow$ *Count* | *EventCount* | `first` | `last`
AttributeName $\longrightarrow$ invoker | date | *String*
AttributeValue $\longrightarrow$ thisUser | *String*
String $\longrightarrow$ { XML CDATA }

Figure 4.8. Syntax for condition definition

by the system, based on the model presented in (Roberts and Verjus, 1977). There are two kinds of system defined events:

1. *Activity events* are related to instantiation and termination of activities.
2. *Operation events* represent execution of role operations.

In the collaboration model, events related to each activity and each role operation are specified with corresponding names, for example `opId` and `activityId`. There are two types of events related to each system defined event, `start` and `finish`, representing the start and finish of an operation or an activity.

Users can generate an *application defined event* as part of an operation. These events are defined to import objects' states in the event space for condition specification. An *application defined event* defines event attribute names, and the values for these attributes are generated at runtime. *Application defined events* are generated

as part of an operation's action and are specified with *NotifyEvent* tag as shown in Figure 4.10.

As presented in Figure 4.8, an `EventName` can be of five types. Multiple occurrences of a given event type — such as multiple executions of an operation — are represented by a list. A *list* operator, `( )`, returns all the events of the specified type. For example, `(EventName)` lists all the events of type `EventName`.

The *count* operator, `#`, returns the number of occurrences of a given event type. e.g., `#(EventName)` returns the the number of occurrence of the event *EventName*. An index i in the event-list, expressed as `EventName(i)`, returns the i'th occurrence of specified type of the event. The index *first* and *last* represent the first and the last occurrence of the event, respectively.

One can also specify a derived event type by filtering an event list based on some predicate on the event's attributes. For each activity and operation events, there are two predefined attribute-names: *invoker* and *date*. For example, for a role operation execution, using the expression `opId.start(invoker=thisUser)`, we can define a filter based on the operation invoker identity. The expression `#(opId.start(invoker=thisUser))` counts the number of times the currently executing user has invoked this operation.

When an operation is requested and the operation's precondition is satisfied, the operation's *start* event is generated. In the event model, the precondition-check for an operation and the generation of the corresponding operation's *start* event is atomic. An operation's *finish* event is generated either implicitly or explicitly depending on the actions that are performed as part of the operation. If the operation action does not have any grant permissions for object access, the operation's *finish* event is generated implicitly. Otherwise, the operation invoker explicitly generates the *finish* event by terminating the interaction session initiated through the operation [1].

[1]Models of interactions among users, roles, and objects and corresponding sessions, initiated through

As presented in Figure 4.8, in the specification model, conditions are specified as predicates on events. These predicates are expressed in two ways:

1. *EventCount Relation Count*
   E.g.the predicate, `#opId.start>0`, is true when the operation `opId.start` is executed at lease once.

2. *EventName*(*Index*)[AttributeName=AttributeValue]
   E.g. the predicate, `opId.start(last)(invoker = thisUser)`, is true if the invoker of the last operation *opId.start* is same as the current invoker.

When the operation *opId* instantiates an activity named *activityId*, there are four events corresponding to the lifecycle of this operation: (1) `opId.start`, (2) `activityId.start`, (3) `opId.finish`, and (4) `activityId.finish`, in this order. Here, `opId.start` imply that precondition, if any, is satisfied and the operation has started, both `activityId.start` and `opId.finish` imply that an instance of the *activityId* has started, and `activityId.finish` implies an instance of the *activityId* has terminated.

4.4.5. Role Specification

As shown in Figure 4.9, a role specification includes role name, *reflected* roles if any, role admission, validation, and activation constraints, role operations with their preconditions, and can have an owner.

The specification model supports assignments of members in a role from a parent activity to role in a child activity. There are two mechanisms that support such assignment of role members. The first mechanism, *static role assignment* or *role reflection*, is specified using the *Reflect* construct. In role reflection, all members of specified roles in the parent activity become members of a role in the child activity when that activity is created. Removal of a member from the reflected role (i.e. the

operations, are discussed in Chapter 6

role in the parent activity) also implies removal from the role in the child activity.

In Figure 4.2, members of the *Instructor* and the *Assistant* roles are admitted or assigned to the *Grader* role in an instance of the *Examination* activity, when the activity instance is created. Similarly, all members of the *Student* role of a course are admitted to the *Examinee* role in any nested examination activities. The corresponding specification for static role assignments in Figure 4.2 is presented in Figure 4.6.

The second mechanism is *dynamic role assignment* in which role admission and validation constraints specify the members that can be admitted to the role at runtime. *Dynamic role assignment* can have an additional requirement in which users for role admission are specified at the time of activity instantiation. As show in Figure 4.5, the template definition uses the *AssignedRoles* tag to specify roles for which some member must be assigned at the time of activity instantiation.

As shown in Figure 4.6, members of the *Instructor*, *Assistant*, and *Student* roles have to be assigned at the time of instantiation of a *Course* activity. Similarly, members of the *Examiner* and *Candidate* roles have to be assigned at the time of instantiating an *Examination* and an *ExamSession* activity, respectively. As shown Figure 4.2, based on role constraints a member of the *Instructor* role can be a member of the *Examiner* role, a member of the *Grader* role can be a member of the *Checker* role, and a member of the *Examinee* role can be a member of the *Candidate* role.

Role admission is discussed in Sections 4.4.7 and 4.4.9. Meta administrative policies related to *Owner* tags is presented in Section 4.5.

4.4.6. Operation Specification

An operation specification includes a name, and may include a *precondition* and an *action*, as shown in Figure 4.10. The precondition must be true when the operation is invoked. The preconditions associated with operations allow one to specify

RoleDef $\longrightarrow$ **Role** roleId {**Reflect** roleId} [**Owner** roleId]
[**AdmissionConstraints** *Condition*]
[**ValidationConstraints** *RoleCondition*]
[**ActivationConstraints** *Condition*]
{*OperationDef*}

Figure 4.9. Syntax for role definition

coordination constraints as well as various dynamic security requirements, such as condition-based access control and dynamic "separation of duties".

The action part of an operation can assign permissions to the role operation, create a new object or a nested activity, invoke a method on an object, change ownership of an object, and generate *application defined events* for condition specification. Permissions are represented with object ids and corresponding method signatures [2]

Roles can create only predefined types of objects, specified with a codebase, as defined with *NewObjectDef* in Figure 4.10. In the following example, the *InitiateBoard* operation creates a *BulletinBoard* object and grants *read* and *write* privileges to the invoker of the operation.

Example 1: Operation creates new object and grants permissions.

```
Operation InitiateBoard { ...
  Action { board=new Object(BulletinBoard);
           Grant board.read; board.write }}
```

Similarly, roles can only instantiate an activity template, which is in immediate

[2]Other than the specified operations, there are four predefined operations for each role, *Join*, *Leave*, *Admit*, and *Remove*, to manage role membership. However, these operations are not part of the specification model. The role membership related conditions are specified using the `member` and `members` functions.

OperationDef ⟶ **Operation** opId [**PreCondition** *Condition*]
[**Action** actionDef]
ActionDef ⟶ {**Grant** *Permission*}
{*NewObjectDef*}
[*NewActivityDef*]
[**InvokeMethod** objId methodSignature methodParameter]
{**ChangeOwner** objId RoleId}
[**NotifyEvent** appDefinedEventId {*AttributeName*=*Attribute Value*}]
Permission ⟶ objId methodSignature
NewObjectDef ⟶ objId = **new Object** codebase
NewActivityDef ⟶ activityId = **new Activity** templateId {**PassedObject** objId}
{**MemberAssignment** roleId = userId {userId} }

Figure 4.10. Syntax for role operation definition

nesting of the current activity, specified with a *templateId*.

4.4.7. Role Admission Constraints

These constraints control a user's admission to a role to meet the role related security requirements. The admission constraints can be based on role membership cardinality specifying the maximum number of users that can join the role; events that must happen before a user could be admitted in a role, such as a predefined time period, tasks performed by others; previous qualifications requiring that the requesting user is currently admitted in some other given roles; or separation-of-duties requiring that the requesting user is currently not admitted in some other given roles.

For example, the *Assistant* role in Figure 4.2 has the following security requirements. These are selected to illustrate the aspects of security requirements that are expressed using role admission constraints. The following three requirements are expressed using admission constraints in the *Assistant* role in our example *Course* activity specification in Figure 4.11.

Example 2: The following admission constraint specifies a role cardinality constraint for the *Assistant* role ensuring that the member count for this role cannot exceed one.

```
#members(thisRole) < 1
```

Example 3: *Static separation of duty* requirement that the assistant cannot be a student in the course is enforced by the following admission constraint in the *Assistant* role. Here, a member of the *Student* role cannot join the *Assistant* role.

```
!member(thisUser, Student)
```

Example 4: A user is admitted to the *Assistant* role only when at least one member is present in the *Instructor* role is enforced by the following admission constraint.

```
#members(Instructor) > 0
```

In the specification of Figure 4.11, all the role constraints for the *Course* activity is presented. In the figure, admission constraints of the *Assistant* and *Student* roles ensure that they can never have a common member. Similarly, by expressing constraints based on role membership, the *Instructor* and *Student* roles cannot have a common member. On the other hand, the admission constraints of the *Assistant* role allow an instructor to join the *Assistant* role, though an assistant cannot join the *Instructor* role. An exam-session activity contains the roles *Candidate* and *Checker*. Only the examinee creating this activity can be admitted to the *Candidate* role.

4.4.8. Role Validation Constraints

Validation constraints are specified using role membership conditions that need to be valid during the whole lifecycle of a role for a user to be a role member. A validation constraint differs from an admission constraint as follow: (1) It needs to be satisfied not only when a user is admitted to a role but also when a member initiates

```
ActivityTemplate Course ... {
    Role Assistant{
        AdmissionConstraints
            #members(thisRole) < 1
            ∧ !member(thisUser, Student)
            ∧ !member(thisUser, Instructor)
            ∧ #members(Instructor) > 0
    ...}
    Role Instructor {
        AdmissionConstraints
            !member(thisUser, Student)
            ∧ !member(thisUser, Assistant)
    ...}
    Role Student {
        AdmissionConstraints
            !member(thisUser,Instructor)
            ∧ !member(thisUser, Assistant)
    ...}

    ActivityTemplate Examination ... {
        Role Examiner {
            AdmissionConstraints
                member(thisUser, parentActivity.Instructor)
        ... }
        Role Approver {
            ValidationConstraints
                !member(thisUser, parentActivity.Assistant) ∧ !member(thisUser, parentActivity.Student)
        ... }
        Role Examinee (Reflect parentActivity.Student) {
        .... }
        Role Grader(Reflect parentActivity.Assistant, parentActivity.Instructor){
            ValidationConstraints !member(thisUser, Approver)
        }

        ActivityTemplate ExamSession ... {
            Role Candidate {
                AdmissionConstraints
                    member(thisUser, parentActivity.Examinee)
                    ∧ member(thisUser, thisActivity.Creator)
                    ∧ #members(thisRole)<1
                ActivationConstraints
                    date>DATE(May, 10, 2003, 9:00) ∧date<DATE(May, 10, 2003, 11:00)
            ... }
            Role Checker {
                AdmissionConstraints
                    #members(thisRole)<1 ∧ member(thisUser, parentActivity.Grader)
            ...}
        }
    }
}
```

Figure 4.11. Skeleton specification of Course activity template with role constraints using psuedo code

any role operations; and (2) It is also verified when any membership related query is invoked on the role. The above two rules ensure that a user's membership to a role is revoked when the validation constraints are not satisfied.

In the following example, a snippet of the specification in Figure 4.11, a validation constraint is specified for the *Grader* role based on users' membership in the *Approver* role. This ensures that when a user can join the *Grader* role or performs any operation in the *Grader* role, the user is not a member of the *Approver* role.

Example 5: Validation constraints.

```
Role Grader {
  ValidationConstraints !member(thisUser, Approver)
}
Role Approver {
  ValidationConstraints
  !member(thisUser, parentActivity.Assistant)
  ∧ !member(thisUser, parentActivity.Student) }
```

Moreover, the *Approver* role has validation constraints. This in turn ensures that when a user joins the *Grader* role or performs any operation in the *Grader* role, the user is also not a member of the *Assistant* or the *Student* role.

4.4.9. Role Admission on Activity Creation

As described earlier, the specification model supports two mechanisms to admit members of roles in the parent activity to roles in a child activity. In the first mechanism, as shown in Figure 4.2, using role reflection, all members of the *Student* role in the parent activity are admitted to the *Examinee* role, subject to the *Examinee* role's admission constraints. This specification is shown below.

Example 6: Role reflection

```
Role Examinee (Reflect parentActivity.Student)
```

In the second mechanism, *dynamic role assignment*, in which users for role admission are specified at the time of activity instantiation. The template definition uses the *AssignedRoles* tag to specify roles for which some member must be assigned at the time of activity instantiation, as specified for the *Candidate* role of the *ExamSession* activity.

Example 7: Activity template definition requiring role member assignment.

```
ActivityTemplate ExamSession( ExamPaper exam, AssignedRoles Candidate)
```

Below is the partial specification of the *StartExam* operation of an *Examinee* role in the *Examination* activity template. When an examinee invokes this *StartExam* operation, an instance of the *ExamSession* template is created and the participant creating the instance is dynamically assigned to the *Candidate* role.

Example 8: Creation of activity instance based on template definition.

```
Operation StartExam { ...
  Action { session=new Activity ExamSession(exam, Candidate=thisUser)}}
```

4.4.10. Precondition and Action Specification

A user can perform an operation's action only when the precondition for the operation is true. The preconditions associated with operations allow one to specify coordination constraints as well as various dynamic security requirements, such as condition-based access control, dynamic "separation of duties" and context-based access control.

As presented in Figure 4.10, the action part of an operation may grant permissions on objects, or create a new object or a nested activity. If the action part is empty, then the operation is used primarily for coordination purposes using its *start*

and *finish* events. A keyword *new* is reserved for specifying creation of an object or activity. In the following example of operation specification, the *Examiner* role in the *Examination* activity has the operation *SetPaper*. The operation *SetPaper* can be performed only once as specified by the precondition. This operation results in creation of an *exam* object of type *ExamPaper* and an invocation of the *setQuestions* method of this object.

Example 9: Operation creates object and grant permissions. Precondition for intra-role coordination policy: *cooperative participation model*

```
Role Examiner {
  Operation SetPaper {
    Precondition #(SetPaper.start) = 0
    Action { exam=new Object(ExamPaper);Grant exam.setQuestions(data) }}
```

Preconditions also enable us to specify coordination constraints, for both *inter-role* and *intra-role* coordination. For example, in Figure 4.2, a student in the *Examinee* role cannot execute the *StartExam* operation until the *Approver* has approved the exam paper.

Example 10: Precondition for intra-role coordination policy: *independent participation model.*

```
Operation StartExam {
  Precondition #(Approver.ApprovePaper.finish)=1
               ∧ #(Examinee.StartExam.start(invoker=thisUser))=0
  Action session=new Activity ExamSession( exam, Candidate=thisUser)}
```

The precondition for this operation also illustrates an intra-role coordination policy, which allows each member in the *Examinee* role to independently start an exam session. This illustrates the *independent participation model* for the members in the *Examinee* role. In the previous example snippet, the precondition of the *SetPaper*

operation in the *Examiner* role illustrates the *cooperative participation model* where only one of the role members can execute the *SetPaper* operation.

In our example, an *operational separation of duty* constraint is specified for the *Approver* role. An examiner may prepare an exam-paper and an approver can approve the paper, but the approver should not be able to approve an exam-paper that he or she has prepared. This specification is shown below.

Example 11: Precondition for *operational separation of duty.*

```
Role Approver {
  Operation ApprovePaper {
    Precondition #(SetPaper.finish)=1
                 ∧ #(SetPaper.finish(invoker=thisUser))=0 } }
```

Similarly, in an office system, a manager may prepare an invoice by invoking *PrepareInvoice* operation and approve an invoice by invoking *ApproveInvoice* operation, but should not be able to approve his/her own invoice. The specification shown below is an example of an *object based separation of duty* policy.

Example 12: Precondition for *object based separation of duty.*

```
  Operation ApproveInvoice
    Precondition #(PrepareInvoice.finish(invoker=thisUser))=0
    Action /* approve the invoice */
```

A skeleton specification of the *Course* activity template, with operation and owner specification is shown in Figure 4.12

4.4.11. Role Activation Constraints

These are used to specify common preconditions for all operations defined for the role. Every time a user performs a role operation the activation constraints, as well as

```
ActivityTemplate Course ....
    Role Assistant ....
        Operation AccessBoard {
            Precondition #(InitiateBoard.start)=1
            Action {Grant board.read; board.write }}
    Role Instructor ....
        Operation InitiateBoard {
            Precondition #(InitiateBoard.start)=0
            Action { board=new Object(BulletinBoard);
                     Grant board.read; board.write }}
        Operation StartExamination {
            Action exam=new Activity Examination(Examiner={})}
    Role Student ....
        Operation AccessBoard {
            Precondition #(InitiateBoard.start)=1
            Action {Grant board.read; board.write }}

    ActivityTemplate Examination (Owner Instructor.....
        TerminationCondition #(exam_session.finish)=#members(Examinee)
        Role Examiner ...
            Operation SetPaper {
                Precondition #(SetPaper.start)=0
                Action { exam=new Object(ExamPaper); Grant exam.setQuestions(data) }}
        Role Approver (Owner Adm2) ...
            Operation ApprovePaper {
                Precondition #(SetPaper.finish)=1 ∧ #(SetPaper.finish(invoker=thisUser))=0 }}
        Role Examinee ....
            Operation StartExam {
                Precondition #(ApprovePaper.finish)=1 ∧ #(StartExam.start(invoker=thisUser))=0
                Action session=new Activity ExamSession( exam, Candidate=thisUser)}
        Role Grader ...

        ActivityTemplate ExamSession Owner Creator .....  {
            TerminationCondition #(Checker.Grade.finish)>0
            Role Candidate ...
                Operation OpenExam{
                    Precondition #(OpenExam.start)=0
                    Action { ans=new OBJECT(AnswerBook);Grant exam.readPaper() }
                Operation Write {
                    Precondition #(OpenExam.finish)>0
                    Action Grant ans.writeAnswer(data) }
                Operation Submit {
                    Precondition #(Write.finish)>0
                    Action ChangeOwner(ans, Checker) }
            Role Checker ...
                Operation Grade {
                    Precondition #(Candidate.Submit.finish)=1
                    Action Grant ans.setGrade(data)}
}}}
```

Figure 4.12. Skeleton specification of Course activity template with operations using psuedo code

validation constraints are checked. As opposed to validation constraints, activation constraints are not validated when membership information is queried.

The following example presents an activation constraint, where the candidate can perform an operation only during the designated time for the exam. In this example of *Candidate* role, the activation constraint is not validated when any other role query the *Candidate* role's membership information.

Example 13: Role activation constraints.

```
Role Candidate { ....
  ActivationConstraints
    date>DATE(May, 10, 2003, 9:00) ∧date<DATE(May, 10, 2003, 11:00)
...}
```

A *dynamic separation of duty* constraint, such as a user should not perform operations in two roles, needs to be specified as part of role activation or validation constraints as oppose to role admission constraints. Role admission constraints are only checked when a user is admitted to a role, not when the user performs a role operation. Similarly, *previous qualifications* that must be ensured during role operation invocation need to be specified as activation or validation constraints.

A cardinality constraint, which specifies a minimum number of members that must be present before any role operation can be performed, are specified as activation constraints. In the following example, we present activation constraints for a *CodeReviewer* role of a software development team. A minimum of 3 members must be present for the role members to perform any operation. And at least a member from both the *Developer* and the *ProjectManager* must be present during the role activities.

Example 14: Minimum role cardinality constraint.

```
#members(thisRole)>=3
∧ #(members(thisRole) ∩ members(Developer))>0
∧ #(members(thisRole) ∩ members(ProjectManager))>0
```

4.5. Meta Policy Specification

The *Owner* role of an entity – activity instance, role, and object – represents the administrator and its members are responsible to manage the entity by enforcing entity specific policies. Based on the entity specific policies, the owner of a role can admit or remove users in the role, the owner of an object can provide access to the object, and the owner of an activity can terminate the activity. Another meta role *Creator* is defined to handle dynamic aspects of a collaboration specification. The user who instantiates an object or activity is a member of the *Creator* role.

In this model, the collaboration designer can specify an owner for every entity. The membership rules for the *Owner* role are as follows:

1. The template specification may indicate which role would be the owner of an entity. Meta roles, such as *Creator*, can be specified as an owner.

2. If not explicitly specified:

   - for an activity, the owner of the parent activity is the owner;
   - for a role, owner of the activity in which the role is defined becomes its default owner; and
   - the default owner of an object is the role that creates it.

   For the top level activity, the creator is the owner.

To handle dynamic ownership aspect of an object, the *ChangeOwner* primitive is supported. The ownership of an object can only be changed by its current owner.

Figure 4.13 presents the conceptual view of the *Course* activity template in Figure 4.2 (Complete specification for the *Course* activity template can be found in Appendix D). In Figure 4.13, the role *Adm1* is the creator and owner of the top level *Course* activity instances. For any nested *Examination* instances, the *Instructor* role is assigned as the owner. Following the default owner assignment rule, *Instructor*

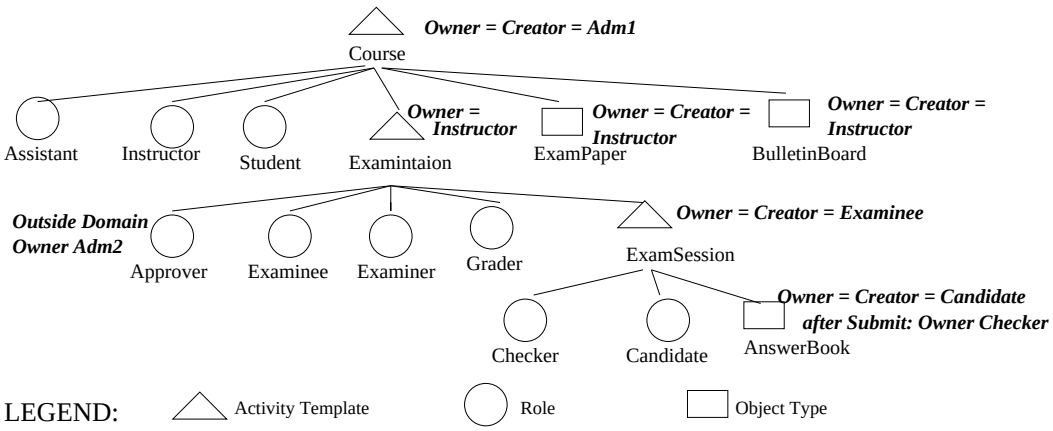

Figure 4.13. Static assignment of owners in an activity template hierarchy

role is the owner of the *Examiner*, *Grader*, *Examinee* roles. The owner of *Approver* is assigned to an outside domain role *Adm2*, showing that it may not always be possible to assign the participants of the initiator domain to manage all the roles and objects. As *Creator* is assigned as owner for the *ExamSession* template, the examinee who initiates an exam-session is the owner of the session. To comply with security requirements, a single entity may not be able to manage an object through its lifecycle. *Candidate* creates and owns the objects of type *AnswerBook*; however, after the execution of the *Submit* operation, the ownership is transferred to *Checker*. The resulting ownership relations are shown in Figure 4.14.

In Figure 4.14, the *Examinee* role is owned by the *Instructor* role. In our ownership model, this implies that members of the *Instructor* role are responsible to enforce any role related policies, specifically role operation preconditions, role admission and validation constraints. Following the ownership relations in Figure 4.14, as the *Examinee* is owned by the *Instructor*, and the *Instructor* is owned by the *Adm1*, the *Examinee* can be owned by the members of *Adm1*. This is possible as the members of the *Adm1* role can have themselves admitted in the *Instructor* role. With trusted assignment of members in owner roles, the *Adm1* is trusted to follow role admission constraints [3].

[3]In case members of multiple roles are required to jointly manage an entity, a new role is declared

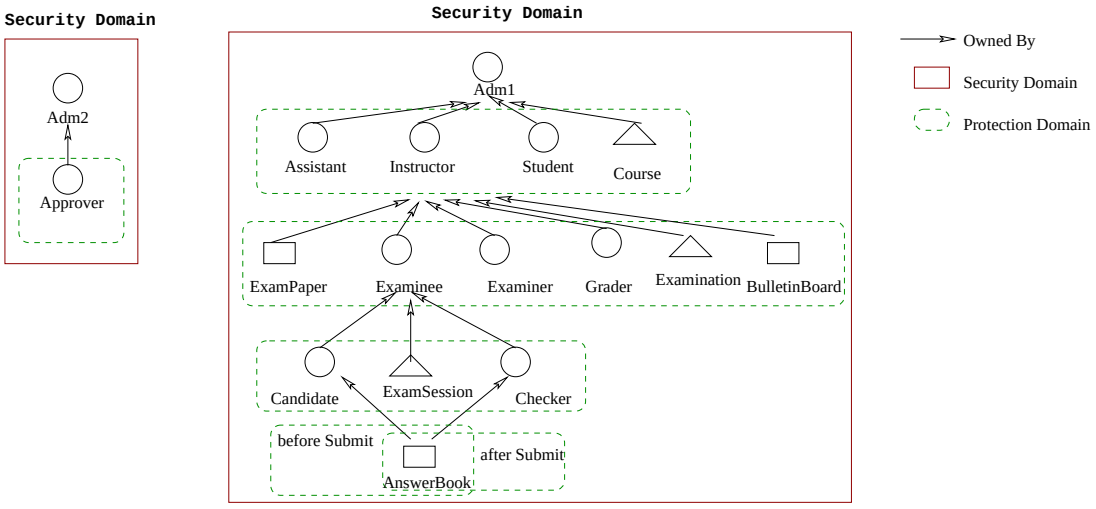

Figure 4.14. Security and protection domains on the owner hierarchies derived from Figure 4.13

4.6. Security and Trust Domains in Decentralized Administration

In distributed systems, trust is viewed as the result of an assessment made by an observer about a person, organization, or any other entity (Denning, 1993). During initiating a secure session, trust is assumed among interacting distributed nodes to correctly perform specific tasks. Based on these specific tasks, trust is classified, such as providing identification of another entity, providing good quality keys, maintaining secrets, maintaining synchronized clocks, following protocol specifications, providing recommendation, and not interfering with other entities' session (Yahalom et al., 1993). In distributed collaboration systems, additional classification of trust is needed for properly manage entities. Based on the aspect of the policies, trust can be classified representing *trust classes*, such as, managing role membership, maintaining coordination consistency, and enforcing coordination policies and security policies. By trusting an entity to enforce a policy, we in turn trust both the owner of the entity and the *administrator* of the policy enforcement mechanism. Due to incompetence or maliciousness of owners or administrators, there may exist potential reasons

with members from those roles and assigned as the owner.

for distrust among participants of a collaboration. Here we define several concepts that are required for proper assignments of owners in our specification model.

- A *security domain* is an administrative or organizational domain, where security policies are enforced under a single administrative entity. Owners within a security domain may form owner hierarchies. In peer-to-peer management of collaboration entities, a security domain can represent a specific participant.

- Within a security domain, an owner is associated with a *protection domain*, which signifies a trusted protection mechanism for enforcing polices. A protection domain also represents the entities owned by an owner. In the execution environment, protection domains are created in *trusted servers*. At runtime each role is assigned a trusted server. A trusted server for a role is not the server where the role itself is managed, but it is the server where the role manages the entities that it owns. Assigning a role member's personal computer as a trusted server may not always work as role members can dynamically join and leave. There are meta policies that govern how a trusted server is selected for a role:

  1. The default rule for selecting a role's trusted server is to select its owner's trusted server.
  2. The collaboration definition can explicitly specify that the role's trusted server must be selected from trusted servers of its participants. Different policies can be employed to assign the trusted server. For example, the server of the first member admitted to a role, a common server from participants' trusted servers, or a server which is trusted by the largest number of participants.

  In Figure 4.14, the entities in ownership hierarchies under *Adm1*, *Adm2*, represent two different security domains. The protection domains associated with the owners *Adm1*, *Adm2*, *Instructor*, *Examinee*, *Candidate*, and *Checker* are also shown in Figure 4.14. In Figure 4.15, the *Adm2* role has selected the trusted server *T1* to manage the entity, i.e., *Approver* role, that it owns. On the other

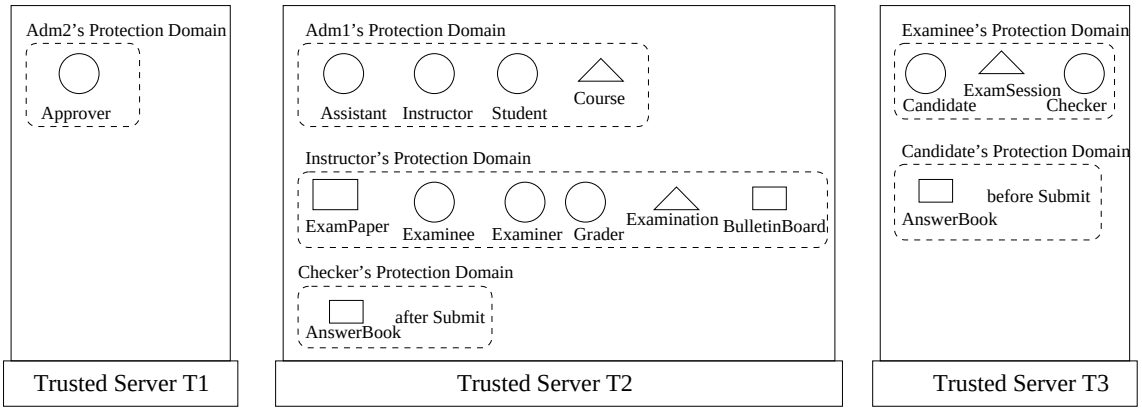

Figure 4.15. Protection domains on the the trusted servers selected by the owners in Figure 4.14

hand, the owner roles *Adm1*, *Instructor*, and *Checker* are assigned trusted server *T2* and the owner role *Examinee* and *Candidate* are assigned trusted server *T3* to manage the protection domains under their ownerships as shown in Figure 4.14.

- A distributed collaboration can span across multiple security domains. There can be a lack of trust among these security domains for proper enforcement of policies. Moreover, within the same security domain, the participants may not trust each other to properly enforce policies. For a specific class of trust, a *trust domain* represents a set of owners that trust each other to correctly enforce certain class of policies. Hence two different security domains can be within the same trust domain. On the other hand, a subset of the owners within different security domains may form a trust domain. Moreover, trust domains in collaboration systems may not be static, that is, the owner initially trusted for a specific trust class, may not be trusted to enforce specific policies as a collaboration activity progresses. In such cases, the owner for the entity may need to be changed during the collaboration.

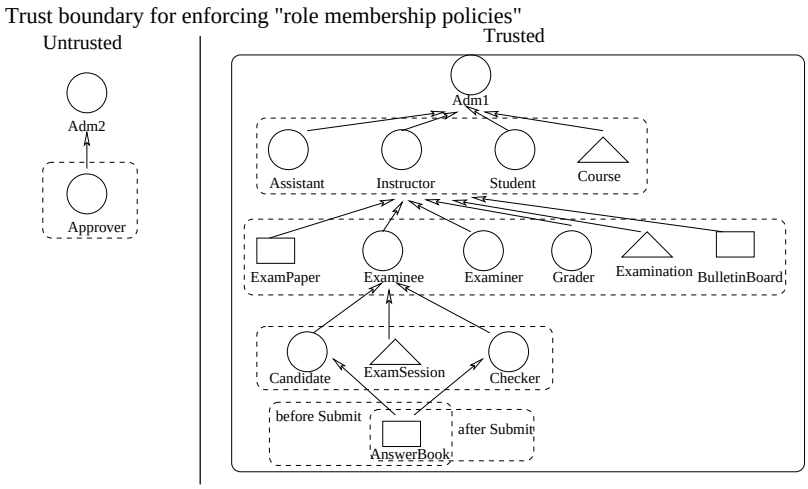

Figure 4.16. Trust domain for enforcing role membership policies

Security domain is an organizational concept that exists before a collaboration is designed. Protection domain represents the execution environment in decentralized management of a collaboration. These concepts require consideration when associating trust classes with participants of the owner roles. Defining trust domains is the responsibility of the collaboration designer. The designer can verify a collaboration specification based on different trust assignments.

In our example, in the security domain with ownership hierarchy under `Adm1`, entities trust their owner roles for proper enforcement of role membership related policies. However, there is a mutual distrust between *Adm1* and *Adm2* role members of the two security domains. For verification purpose, the entities under the *Adm1* owner hierarchy forms a trust domain for enforcement of role membership related policies, as shown in Figure 4.16. Under the same ownership hierarchy of *Adm1*, participants of the *Student* role are not trusted by others to enforce any object access policies. In Figure 4.17, a trust domain is presented that excludes the entities that can be managed by the participants of the *Student* role. Such assignment of trust domain for entities are utilized to verify security properties when all the users in a collaboration

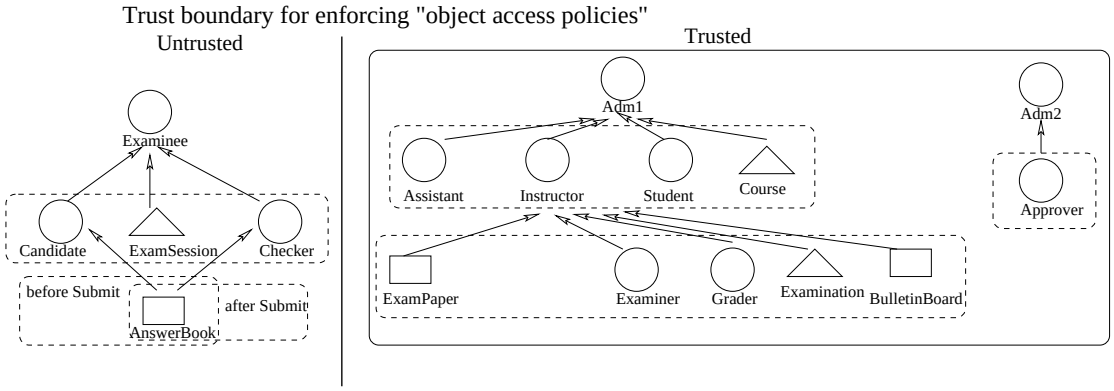

Figure 4.17. Trust domain for enforcing object access policies

cannot be trusted to enforce specific policy. This *trust domain* concept is utilized in our verification methodology presented in the next Chapter.

4.7. Related Work: Collaboration Policy Specification Languages

Various specification techniques are used to specify CSCW tools for various purposes: to formalize and reason about the systems, to prevent deadlocks among coordinating users, to automatically realize the desired system, and to enforcement various policies.

Formal Description Techniques (FDTs) (Lai and Jirachiefpattana, 1998), e.g., extended finite state machine based Estelle and SDL, temporal logic based models, process algebra based LOTOS etc. are widely known for protocol verification of distributed and networked system. Although these FDTs allow reasoning and verification, they are hard to understand, and FDT based specifications are written for system level entities, such as processes. These FDTs are not convenient to specify user level collaboration policies. Specification of a shared drawing tool We-Met in LOTOS is presented in (Rekers and Sprinkhuizen-Kuyper, 1993). The authors mentioned that LOTOS specifications are hard to develop and interpret. LOTOS requires explicit specification of communicating components, and mainly uses synchronization

operators to avoid deadlock or other unintended situations.

Trellis (Furuta and Stotts, 1994) is a colored timed net (CTN), extended form of Petri Net, based collaboration coordination specification facility, which uses a model checking system for verification. A floor control coordination with hyper-text composition is presented based on Trellis. It achieves selective display of events (e.g. buttons associated with a transaction) using attribute/value pair associated with CTN's entities, such as place, transaction, arc, or the whole CTN. Work presented in this thesis in not limited to floor control coordination and the security policies verified takes into account trust issues among collaborating users.

GPSG (Natalie S. Glance and Pareschi, 1996) is a rule based grammar for processes, which uses constraints to express dependencies among related activities and documents. Unlike other process grammars, GPSG is both activity-oriented, and document-oriented and they are duals of each other. Contrast to other workflow system, it triggers tasks based on document states. Several views of the same process can be seen, for example a task view, a time view, or a role view.

AMIGO (Danielson and Pankoke-Babatz, 1988) activity model facilitates predefinition of generic communication process description, termed *Activity*. Formal syntax of Activity including its components, such as roles, message objects, functions, and rules are presented; however, the model is not tested with any prototype implementation. The primary goal of this model is to express the group communication process for coordination.

COSMOS (Dollimore and Wilbur, 1989) supports configurable asynchronous group interactions and provides a script language with concurrency constructs. Based on task analysis, a set of scripts termed activity is defined. However, runtime modification of these scripts and wrong design can result in deadlocks. A primary difference between these research and the research presented in this thesis is that we have developed a role based security model to express a wide range of security requirements in

distributed collaboration systems where collaborating users may not trust each other.

In (Gelernter and Carriero, 1992), significance of coordination language of building asynchronous ensembles, i.e., a collection of asynchronous activities that communicate are discussed. It argues for coordination languages, e.g., Linda, for portability and heterogeneity. Moses (Minsky and Ungureanu, 1997) addresses that Linda like shared-space coordination mechanisms do no scale well. It utilizes a controller interposed between a client and the network. Messages to and from the client are checked against the control information in the controller and can be blocked or modified. Scalability is achieved by keeping the control states in the controller.

CSDL(Cooperative System Design Language) (DePaoli and Tisato, 1994b,a, 1993) is a specification and a design language for synchronous shared-workspace type CSCW applications. It specifies coordination based on membership conditions of users' in roles. For example, in a collaborative drawing system with two roles: *drawer* and *viewer*, a user can join the *drawer* role and draw if no user is present in the drawer role. This specification is mapped with a communication specification specifying which roles are allowed to access which channels. Hence, roles contain specific coordination and access rights, and by declaring exclusive roles, coordination policy is specified. User identities are not used in the specification. The policy is based on a user's membership in roles.

Preliminary works of a CSCW system WORLD and its specification language, Introspect, are presented in (William J. Tolone and Fitzpatrick, 1995). Introspect is composed in three levels: meta-specifications, specifications, and instantiations. The meta-specification level provides the support for radical runtime modification to Introspect specification. Introspect specifies locales and actions, and maps them with a process model.

Ponder (Damianou et al., 2001) is a declarative language designed for specification of enterprise wide security and system management policies. It provides a wide range

of primitives to support from simple authorization policies to difficult meta-policy and obligation policy. As this language tries to cover a broad domain, it borrows a subset of OMG(Object Management Group) initiated Object Constraints Language(OCL) to specify constraints and may require suitable scripting language to specify obligation actions.

Similar to our model, other RBAC models have defined context-sensitive access control constraints based on different concepts for the context of a role, such as *team* in Team Based Access Control (TMAC) (Thomas, 1997), *domain* in role based management (Lupu and Sloman, 1997), *role template* in (Giuri and Iglio, 1997), and *authorization template* in (Huang and Atluri, 1999). Compared to our role based model, none of these models support meta-roles for administrative policies in decentralized management of collaboration systems.

4.8. Summary

In this chapter, we have presented a specification model for security and coordination policies in collaboration systems. The model is based on two main concepts: role and activity. The syntax for the specification model as well as example specifications are presented. The model can specify requirements related to context sensitive security polices and decentralized administrative policies. Based on a *Course* collaboration example and other short examples, we have shown how different security policies and role constraints can be specified in our model. In the next chapter, we will discuss how a specification can be verified against a set of coordination and security requirements, using the *Course* example. Lastly, the security domain, protection domain, and trust domain are defined. These concepts are utilized for design of our policy driven distributed middleware as well as for verification of security requirements in decentralized administration of collaboration systems.

Chapter 5

Static Verification

In this chapter, we present a methodology for verification of security policy specification using model checking in distributed CSCW systems. In the previous chapter, we have presented a role based specification model for security and coordination policies based on the unique security requirements in decentralized CSCW systems. We have also shown how this specification model can express dynamic security requirements in collaboration environments. Given global security requirements, verification of a specification needs to ensure completeness and consistency of the specification. The goal of our methodology is to ensure that sensitive security requirements cannot be violated when a collaboration involves multiple security domains and the participants of the collaboration cannot be trusted.

As part of the verification methodology, we present how finite-state techniques, such as model checking, can be utilized for correctness verification in a role based collaboration model. A motivation of our approach is to use an existing model checking tool, SPIN (Holzmann, 2003), to verify security requirements of distributed CSCW systems during the design phase. A problem in automata based verification models is state space explosion, i.e., the exponential increase of state size with addition of new system components. In this chapter, we present the techniques that we have developed to handle the state space explosion problem. Another challenge is the development of verification models to check security constraints, which may be inter-dependent.

Modification of the specification to comply with a requirement may violate any of the previously verified requirements. Our verification methodology ensures how different aspects of security requirements, such as task-flow, role constraints, information flow, assignment of administrative roles, can be verified. Based on these requirements, we have developed several interrelated verification models for the model checker SPIN. For a given specification of a collaboration expressed in XML, these models are constructed from the specification.

The primary purpose of this chapter is twofold. First, the role based model can specify security and coordination policies for distributed collaboration systems. Second, a verification model, which ensures that the specification satisfies security requirements (Gunter et al., 2000). The focus of the verification process is on a methodology for security policy specification including administrative policies to ensure that sensitive security requirements cannot be violated by untrusted participants in distributed management of collaboration systems.

In the next section, we present different aspects of coordination and security requirements that a collaboration designer can specify as global properties during designing a collaboration specification. In Section 5.3, the challenges of model checking for coordination and security requirements as well as the verification methodology are presented.

5.1. Global Properties for Verification

Verification of a collaboration specification has two distinct motivations. First, it has to ensure that the specification is not incomplete or inconsistent. Second, it has to ensure that global security requirements are satisfied by the specification. Correctness and consistency of task and role modeling and their corresponding specification are often weaved with verification of global security requirements.

5.1.1. Reachability of Operations

A primary correctness requirement is related to liveness properties that each of the role operations can be executed, i.e., they are reachable. An operation in our model is unreachable if its precondition can never be satisfied. In the following example, the two inter-dependent role operations represent a deadlock, where none of the operations can be performed.

Operation op1 **Precondition** #(Op2.finish) = 1
Operation op2 **Precondition** #(Op1.finish) = 1

5.1.2. Task Flow

Task modeling is an integral part of collaboration specification in our role model. In our specification model, coordination policies including task synchronization and task-flow constraints are specified with roles as operation preconditions. Such specifications are unable to provide the global view of task and activity dependencies to a collaboration designer. The designer may like to verify task-flow requirements independent of other role constraints, with an alternative form of expression. To facilitate such checks during the design, task-flow requirements can be expressed using *path expression* (Campbell and Habermann, 1974) constructs, such as `sequence (;)` and `selection (())` with a `count (:n)` restrictor, where `n` can be a constant, or "+" representing one or more and "*" representing unbounded.

The task-flow requirement for the *Examination* activity, as discussed in Section 4.3, is expressed as below which requires that a *SetPaper* operation is performed before the *ApprovePaper* operation, an *ExamSession* activity can be started only after an *ApprovePaper*, and the number of exam-sessions has to be equal to the cardinality of the *Examinee* role before the *Examination* terminates.

Examination := Examiner.SetPaper; Approver.ApprovePaper;
Examinee.ExamSession:#member(Examinee)

5.1.3. Role Based Constraints

Incorrect or inconsistent role based constraints can result due to conflicting role admission/validation constraints. Consider the following example, where a member of role A cannot be a member of role B. On the other hand, Role C's admission constraints require that its member has to be a member of both role A and B when joining C, which cannot be satisfied.

Role B **Validation Constraints** !member(A)
Role C **Admission Constraints** member(A) & member(B)

Though there are several role constraints specified for the example in Figure 4.11, we choose the following two role related requirements to be verified.

RC1 *A member of the checker role can never be a candidate.*

RC2 *The student who initiates an exam-session should be the only one who joins the candidate role.*

5.1.4. Information Flow and Confidentiality

As RBAC is "policy neutral", we can model information flow constraints by classifying roles with disjoint members with implicit labels. By doing so, a collaboration designer may like to verify if such constraints can be satisfied. Similarly, constraints can be specified that information can flow only after certain conditions are satisfied, or certain information cannot flow to specific roles.

In our course activity example, the designer intends to enforce and verify the following two confidentiality requirements.

IF3 *A member of the examinee role cannot access the content of the question paper before start of his/her own exam session.*

IF4 *Identity of a candidate should not be known to the member of the assistant role who grades the candidate's answer book, before the submission of the grades.*

5.1.5. Integrity and Access Leakage

Due to owner assignments to roles and objects, members of the owner roles can get extended privileges. Collaboration designer can express integrity policies as global properties to check that access rights are not unintentionally leaked during owner assignments. In our example, the collaboration designer specifies the following requirement.

AL5 *A checker should not grade an answer-book if the exam-paper is not approved by the approver.*

5.2. Finite-State Based Model Checking for Security Policies

SPIN (Holzmann, 2003) is a model checker based on an automata theoretic approach (Vardi and Wolper, 1986). In SPIN, a model of a system to be verified is specified in PROMELA (a Process Meta Language). The desired system property can be expressed in LTL (Linear Temporal Logic) using temporal operators `always` (□), `eventually` (◇), and `until` (`U`).

Given the model of a system and a desired property of the system, SPIN converts the model of a system and the negation of a desired system property to finite Buchi automata, which accepts infinite words thus representing non-deterministic behavior of reactive systems (Vardi and Wolper, 1986). Next, SPIN generates a language intersection of these two automata and uses efficient depth-first search algorithm with a partial order reduction method to find a trace of the counter-example for the desired property (Holzmann, 2003).

In our approach, the collaboration specification in XML is manually converted to PROMELA. However, additional components are needed to be added to the model to verify properties related to information flow and owner assignments. The search space of such a model, even for a small collaboration specification, can be very large.

5.2.1. Challenges in Model Checking for Security Policies

There are several challenges relevant to finite-state based model checking. First, to overcome the difficulty of state space explosion, i.e. exponential growth of state space when new components are added to the verification model, we exploited various abstraction techniques in the verification model. A system model with all its properties intact produces a large search space. Some of the properties that are not in concern when verifying a specific property can be independently verified. For example, in our verification model for role constraints, to verify users' admission to roles, modeling of role operations that cannot affect users movement among roles is not required. Properties related to such role operations are viewed as independent aspects and verified separately. Also, the model is composed incrementally, adding a new property to the model, given a set of properties that have been verified. The search space of the verification models is reduced by tailoring property specific information. For example, if verification of a property is related to any user's invocation of a method, it is not required for the model to maintain all the users' identities, but rather maintain a bit variable signifying the fact that the user has invoked the method. Abstraction of internal data structure also reduces state space.

Second, for easy translation of the specification to verification model, we treat collaboration specifications as executable. As the verification model resembles the runtime system, the interacting participants' identities need to be modeled as part of the verification model. Due to search space problem of finite state based verification techniques, we need a bound on the number of participants for a verification model. However, for a specification, all roles may not have maximum membership cardinality constraints. Any number of activity instances may be created, which results in an unbounded number of roles. Additionally, there may be no bound on the number of times a role member can access objects or create objects, thus creating an unbounded number of channels through which information can leak.

A collaboration specification can be verified based on a user-defined bound on the number of the users. However, if a specification is verified based on a user-defined

bound, it does not guarantee that the requirements will be satisfied with a larger number of users. In Section 5.4, our goal is to find a bound on the number of users per role whose unique identities are sufficient to verify a given set of requirements for a specification. A bounded number of participants implies that if a requirement is not violated with that number of participants, it will not be violated with a larger number of participants.

The third problem is the inter-weaving aspects of the verification requirements. Among the aspects of the properties presented in the previous section, reachability of operations, task-flow, and role based constraints are correctness properties. On the other hand, confidentiality and access leakage are security properties. The verification methodology needs to ensure that it follows a precedence among the properties it checks. Consequently, we have developed, based on the aspects of global requirements to be checked, four models for efficient verification. The models are termed *Task Model*, *Role Model*, *Information Flow Model*, *Owner Assignment Model*. These models, respectively, verify correctness of coordination constraints, role constraints, confidentiality requirements with and without assignment of administrative roles, and integrity requirements when owners are assigned.

5.3. Verification Methodology

Followings are the steps for verification of a collaboration specification given a set of requirements:

- [Step 1:] Verify the correctness of coordination and task-flow requirements with the *Task Model*. In case of an inconsistent requirement, modify the requirements related to task-flow and repeat Step 1.

- [Step 2:] Verify the correctness of role constraints with the *Role Model*. Modify requirements related to role constraints when the verification fails and repeat Step 2.

- [Step 3:] Verify the confidentiality requirements without assignment of administrative roles using the *Information Flow Model*. Modify constraints in the specification when the verification fails. In case of modification of a role constraint repeat Step 2 and 3. For modification of a coordination constraint repeat Step 1, 2 and 3.

- [Step 4:] Verify requirements related to role constraints, information flow and access leakage using the *Owner Model*. This step in discussed in a later section.

5.3.1. Verification Model for Coordination Requirements

The *task model* is designed to verify aspects related to coordination requirements, such as reachability of operations and task-flow. An exhaustive verification run on this model will report unreachable code, pointing out the operations, which are unreachable. Operations that do not have preconditions and on which no other operation depends are not represented in this model. Moreover, this verification model is not concerned with any requirements related to users' presence in roles, such as "operational separation of duties".

```
01 proctype ExamSession_Activity( ) {
02   bit Write_finish=0, OpenExam_start=0, OpenExam_finish=0, Submit_finish=0, Grade_finish=0;
03     do
04     ::Grade_finish == 0 ->
05         if
06         /* OpenExam */
07         ::  atomic { OpenExam_start == 0 -> OpenExam_start = 1; } OpenExam_finish = 1;
08         /* Write */
09         ::  OpenExam_finish != 0 -> Write_finish = 1;
10         /* Submit */
11         ::  Write_finish != 0 -> Submit_finish = 1;
12         /* Grade */
13         ::Submit_finish == 1 -> Grade_finish = 1;
14         fi
15     ::  Grade_finish != 0 -> ExamSession_finish++; break;
16     od 17}
```

Figure 5.1. SPIN model in PROMELA for ExamSession activity (Task Model)

In Figure 5.1, the task model of *ExamSession* activity specification, as presented in Figure 4.12, after converting to PROMELA is shown. In this verification model, each activity is a process (line 01) and multiple instances of the process can be created. Within a process each operation's precondition is modeled as a guarded statement (line 07, 09, 11, and 13). When the guard becomes true, the following statements after the arrow ($->$) can be executed in a non-deterministic step. The *atomic* statement (line 07) ensures that the precondition check and generation of corresponding *OpenExam_start* event is performed in a single step. The process of the *ExamSession* loops till the termination condition is satisfied (line 15). When the termination condition is satisfied the global variable *ExamSession_finish* is incremented (line 15).

The path expressions for the task-flow requirements are converted to LTL expressions[1]. The path expression for the *Examination* activity will be converted to the following LTL expressions, where *count(event, n)* signifies a true value when the count of *event* is equal to n.

$\Box$(`Examination.start` $\rightarrow \Diamond$ `Examiner.SetPaper.start`)
$\Box$(`Examiner.SetPaper.finish` $\rightarrow \Diamond$ `Approver.ApprovePaper.start`)
$\Box$(`Approver.ApprovePaper.finish` $\rightarrow \Diamond$ `Student.ExamSession.start`)
$\Box$(`count(Student.ExamSession.finish, #member(Examinee))` $\rightarrow$
$\Diamond$ `Examination.finish`)

5.3.2. Verification Model for Role Constraints

Primary goal of *role verification model* is to ensure that (1) all roles can have members and (2) role related constraints can be satisfied. To check the user-related properties, the presence of users in roles is modeled. For this, a minimal number for initial user assignment is calculated. A requirement verified with a minimal number of participants is also valid for any larger number of participants. In Section 5.4, we discuss the techniques for finding a minimal number of identities needed for a verification model. Based on our framework, a minimal number for initial users for a

[1]Currently the path expressions are converted manually, however, this can be automated.

Course activity is `5` with `2` in *Student* role, `1` in the *Assistant*, and `2` in the *Instructor* and *Approver* role. For the verification process, we assigned user identity A and B to *Student*, C to *Assistant*, and D and E to *Approver* and *Instructor* role.

Based on the initial members assigned, the exhaustive search by the model checker reports unreachable codes, in turn, pointing to the membership constraints that cannot be satisfied. For a large collaboration, the exhaustive search with a minimal number of participants may not be feasible due to search space explosion. In such cases, role properties can be specified in LTL for faster verification, as it results in searching for only specific counter-examples. In the *Course* activity, with initial assignment of user C to *Assistant* role, the correctness requirement that the *Grader* role will eventually have a member is expressed as the following expression using LTL.

```
□ ◇ member(C, Grader)
```

In our implementation, the verification model maintains a bit vector, where a bit signifies presence of a user in a specific role type. During initialization, we can express which roles and users require tracking. With `member_present` being the bit vector, SPIN LTL property verifier converts the above requirement to the following expression, where j represents the bit corresponding to the C's presence in the *Grader* role.

```
□ ◇ member_present[j]
```

Case Study – Verification of Requirement RC1: To facilitate the designer to express various types of role constraints, conversion functions for role constraints to LTL expressions are provided. For example, the requirement *RC1* of *static separation of duties*, i.e., a member of a *Checker* role cannot be a member of *Candidate* role is converted to following LTL expression. In the verification run, `x` is replaced by user identities, and `r1` and `r2` is replaced with role names.

```
StaticSOD(r1, r2) := !◇ ( member(x, r1) && member(x, r2) )
```

An optimization of this process is to verify based on only possible member of *Checker* role, i.e, *C*. The following expression specifies that eventually there does not exist a state, where *C* is a member of both *Checker* and *Candidate* roles. This requirement was satisfied.

```
StaticSOD(Checker, Candidate) :=
                    !◇ ( member(C, Checker) && member(C, Candidate) )
```

Case Study – Verification of Requirement RC2: Knowing that users *A* and *B* are initial members of the *Student* role, the requirement *RC2* is expressed as below, where *event(event-type, user, activity)* signifies that the *user* has triggered the specified type of event in the *activity*.

```
!◇ ( member(A, Candidate, Es1) && !event(ExamSession_start, A, Es1))
```

The requirement is specified by negating the fact that eventually user *A* is a member of the *Candidate* role without starting the *ExamSession* instance *Es1*. In this expression an activity *Es1* is added to imply that the *ExamSession_start* event and the *Candidate* role are in the same activity instance scope. As users *A* and *B* are added to the *Student* role in non-deterministic steps, checking for either of their identities is sufficient in this verification. This requirement was satisfied based on the role constraints of the *Candidate* role in Figure 4.11.

5.3.3. Verification Model for Information Flow

Several confidentiality properties, such as noninterference, noninference, and nondeducible, have been formalized, and a comparative discussion on types of information flows in these models can be found in (Zakinthinos and Lee, 1997). In our verification model only explicit information flow (Denning, 1982) is captured, which can be summarized by the following two rules:

1. Given objects `o1`, `o2` and subject `s`, which has *read* permission for `o1` at time `t1` and *write* permission for `o2` at time `t2` with `t2` $\geq$ `t1`, then information can flow from `o1` to `o2`, i.e., `o1` $\longrightarrow$ `o2`.

2. Similarly, given `o` is an object and subject `s1` has *write* permission for `o` at time `t1` and subject `s2` has *read* permission for `o` at time `t2` with `t2` $\geq$ `t1`, then information can flow from `s1` to `s2`, i.e., `s1` $\longrightarrow$ `s2`.

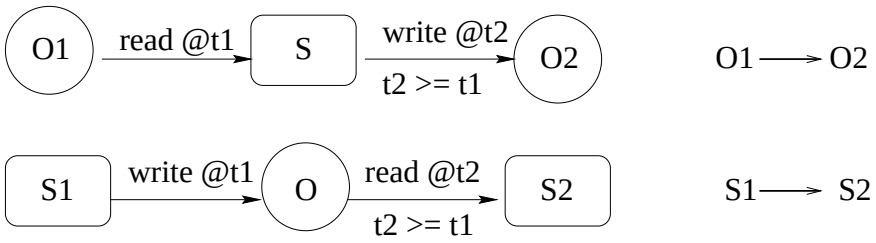

Figure 5.2. Information flow: object to object, subject to subject

To incorporate the above two rules in the model, components related to users' knowledge and objects' internal information are added. In the model, *read* of information is assumed when a method returns any values, and *write* is assumed when any values are passed during method invocation or object creation. One can also rely on explicit declaration of methods in these two categories, read and write, by object designers.

Case Study – Verification of Requirement IF3: Knowing that users A and B are initial members of the *Student* role, we express the information flow requirement *IF3*, in Section 5.1.4, as "user A of the examinee role cannot access the content of the question paper before start of his/her own exam session". This requirement is expressed as below, where *knows(subject, object)* represents a bit in the verification model, which signifies that the *object* content has passed to the *subject* and *event(event-type, user)* signifies that the *user* has triggered the specified type of event.

```
!◇ (knows(A, ExamPaper) && !event(ExamSession_start, A))
```

The requirement is specified by negating the fact that eventually user A knows the content of the *ExamPaper* without starting his/her *ExamSession*. Below we describe the steps through which the original specification was modified to comply with this requirement.

- In our initial run, with initial assignment of users A and B to *Student*, C to *Assistant*, and D and E to *Instructor* and *Approver*, a counter-example was found. In the trace, the *Examiner* leaked the *ExamPaper* to the examinees before the start of their exam-sessions through the *BulletinBoard*. To encode that such an act would not be performed by the *Examiner*, we provided the fact to the model as a tuple *!write(Examiner, ExamPaper, BulletinBoard)*, which meant that *Examiner* would not *write ExamPaper* content to the *BulletinBoard*.

- In the second run, similar situation was found where the *Grader* role leaked the *ExamPaper* through the *BulletinBoard*. However, *Grader* did not have direct access to the *ExamPaper* and the information was leaked from examinee B through grader C to examinee A. A fact that *Grader* would not perform such an act was provided to the model.

- The next verification run found another counter-example showing that examinee B was able to leak the content of the *ExamPaper* through the *BulletinBoard* before user A had started his/her own *ExamSession*. To preserve this property of information flow, we investigated the following two solutions.

- [First Solution:] The *Student* role's privileges on *BulletinBoard* were revoked during the *Examination* activity. This was accomplished by adding a coordination constraint on the operations of the *Student* role accessing the *BulletinBoard*. However, this was very restrictive as the examinee, who had not started an exam-session, was not able to use the bulletin board.

- [Second Solution:] We ensured that the examinee who initiated an exam-session

could not access the *BulletinBoard* during the session. We removed *BulletinBoard* privileges from the *Student* role and added a new role *BulletinBoard-User* in the *Course* activity with *BulletinBoard* privileges. Only by joining this role, a participant of the *Student* role could access the *BulletinBoard*. Additionally, an "operational separation of duties" constraint was specified that an examinee had to leave the *BulletinBoard-User* role during his/her *ExamSession* activity.

- The verification of the confidentiality property failed as examinee A was able to wait for completion of examinee B's exam-session and was able to share the *ExamPaper* through the *BulletinBoard*.

- To prevent this, we added a constraint that the exam-session could only be started after a predefined time, *ExamStartTime*. Moreover, the submit of the *AnswerBook* could not be performed for a period *ExamJoinDuration*, during which other examinees could initiate their exam-sessions. To include these restrictions, we required real time events. However, SPIN does not support verification related to real time. As the proper duration of these events were not necessary for verification, we modeled two events, which signified the events *ExamStartTime* and *ExamJoinDuration*. These events were modeled based on the *SetPaper* event and the *Submit* event of the first examinee, respectively. Finally, the resulting specification was valid for the requirement *IF3*.

5.3.4. Verification with Owner Assignment

Global security requirements can be violated by an owner due to the extended privileges granted to it even though the owner does not violate any policies it is entrusted with. In the trusted owner model, the owner of a role can view identities of the participants of the role, and the owner of an object can read/modify it without any restriction.

On the other hand, owners may not be trusted and may try to violate security policies by not enforcing or by manipulating policies under their control. Based on

these two cases, we have developed two modes, *trusted owner* and *untrusted owner*, for verifying properties under owner assignment model.

For modeling of untrusted owners, we rely on types of policies that can be violated by the owners. The owner of a role can violate *role admission and validation* policies. The owner of an object can violate *object creation* and *dynamic access control* policies by violating operation preconditions. Similarly, the activity owner can violate *activity instance creation policy*, which ensures that only authorized entities can instantiate an activity. All these policies are based on events related to operation invocation and role membership. This requires that events are sent to only authorized subscribers and received only from valid notifiers. In untrusted owner model, entities owned by the untrusted owners can be *misbehaving*. To derive verification models for misbehaving entities, the violation of above policies is categorized into two types:

1. [Violation of role constraints:] These are (1) role admission and validation constraints are not enforced, i.e., a role owner can put any user in the role, and (2) role membership related query may return invalid information.

2. [Violation of operation preconditions:] This results in violation of coordination and dynamic access control policies. There are cases, when a misbehaving entity influences other entities by manipulating the causal dependency of the policies under its control. This type of violation of global requirements results when preconditions for operations that generate coordination events are not enforced. If a misbehaving entity is the notifier of coordination events, the entity is modeled as generating false coordination events or omitting coordination events.

Verification with Trusted Owner

In trusted owner mode, an owner is trusted to correctly enforce all policies under its control. Information flow is the main concern in verification with trusted owner mode. As owner assignment related concerns can be separately checked, in the previous discussion of information flow model, we have not taken into consideration the

effect of owner assignments.

Case Study – Verification of Requirement IF3 under Trusted Owner Model: In our previous case study, in the next step, we checked and found that the owner assignment configuration in Figure 4.14 satisfied the requirement *IF3*.

Case Study – Verification of Requirement IF4 under Trusted Owner Model: The confidentiality requirement *IF4*, knowing that user C is an initial member of the *Assistant* role, is expressed as below, where In this expression a context *Es1* is added to imply that the *Grade_finish* event and the *Candidate* and *Checker* roles are in the same activity instance scope. Moreover, *members(role, activity)* represents users' identities in the *role* of the *activity*.

```
!◇( !event(Grade_finish, C, Es1) && member(A, Candidate, Es1)
     && member(C, Checker, Es1 ) && knows(C, members(Candidate, Es1)))
```

The requirement is expressed as a negation of the error behavior, that is A is a member of the *Candidate* role and C is a member of the *Checker* role in the same exam-session context, and *Candidate* role member's identity, i.e., A's identity is known to C before the *Grade* operation is performed by C. As users A and B are added to the *Student* role in non-deterministic steps, checking for either of their identities in *C's* knowledge suffices for this verification. A counter example was found where candidate A leaked his/her identity through the AnswerBook object. The fact that *Candidate* would not perform such an action was provided to the model. In the consequent verification, requirement *IF4* was satisfied with *Examinee* as the owner of the *Candidate* role.

Verification with Untrusted Owner

The primary problem in verification of requirements under untrusted owner is to identify the entities that can be owned by untrusted participants. For a given requirement, the collaboration designer is responsible to define a trust domain partitioning the participants into two classes: trusted and untrusted. The designer marks a set of

roles which may not be trusted for a set of requirements. Once the initial set of roles with untrusted users are marked, the next step is to find all the roles that untrusted users would be able to join. Among these roles, which can have untrusted users, a subset may be owners of certain other entities. Such an entity is called *potentially misbehaving* as it can be owned by an untrusted user, who can potentially violate policies associated with it. When verified, if this misbehaving entity violates a sensitive global security requirement, it is called a *consequently misbehaving* entity. The goal of our verification process is to identify *consequently misbehaving* entities among *potentially misbehaving* entities.

Followings are the steps that a designer has to perform to verify each desired requirement given a list of potentially misbehaving entities.

1. Within the derived list of potentially misbehaving entities, an entity in the scope of the inner most activity template of the nested activity definition is selected and modeled as misbehaving. As mentioned earlier, an entity is modeled as misbehaving by either (1) violating role constraints or (2) violating operation preconditions.

2. If this misbehaving entity violates a sensitive global security requirement, it is a *consequently misbehaving* entity. Then it is either marked to be managed by a trusted role or the specification is modified to ensure that such a requirement cannot be violated. In the second case, based on the type of modification, the specification has to be checked for consistency using task, role, or information flow verification models.

3. If the requirement is not violated, this potentially misbehaving entity may violate the requirement when modeled as untrusted with other potentially misbehaving entities. In this step, the next inner most potentially misbehaving entity is selected and added to the model with the previous potentially misbehaving entity or entities.

A subsequent problem for the designer is to choose a requirement among all the global requirements to be verified. The concerns of the verification model suggest the type of requirement to be verified.

Case Study – Verification with Untrusted Owner:
In our example *Course* activity, *Adm2* and *Student* roles were marked to have untrusted users. Based on unrestricted admission of users, potentially misbehaving entities were *Approver*, *ExamSession*, *Candidate*, *Checker*, and *AnswerBook*. Among the above potentially misbehaving entities, the *Checker*, *Candidate* and *AnswerBook* are defined in the inner most *ExamSession* activity template. We chose the *Checker* role as our first potentially misbehaving entity to be verified for violation of role-based constraints. In our sample requirements listed in Section 5.1, the aspect of requirements *RC1*, *RC2* and *IF4*, match with the concern verified by the verification model for violation of role constrains. Additionally, the scope of the entity, which is modeled as misbehaving, provides clues on finding requirements which may be violated, e.g., requirements *RC1* and *IF4* are specified in the scope of the *Checker* role.

Verification of Requirement RC1 with Untrusted Owner for Violation of Role Constraints: The verification of the requirement *RC1* that a member of the *Checker* role can never be a member of a *Candidate* role failed. As the role constraint for the *Checker* role was not enforced, *A* being a member of the *Student* role was able to join the *Checker* role. Considering the requirement *RC1* is sensitive for a course activity, the *Checker* role was marked to be managed by a trusted owner. For the next verification, the verification of requirement *IF4* with the *Checker* role being potentially misbehaving was skipped, as the *Checker* role was already marked as consequently misbehaving. In the next step, the *Candidate* role was designated as potentially misbehaving.

Verification of Requirement RC2 with Untrusted Owner for Violation of Role Constraints: The verification of the requirement *RC2*, "the student who initiates an exam-session should be the only one who joins the exam-session", failed with a trace

of counter-example, where the *Candidate* role had admitted user A to the exam-session when user B had initiated the session. This requirement was sensitive for the course activity, and the *Candidate* role was marked to be consequently misbehaving. In our example requirements, as there were no requirements related to *AnswerBook*, we chose *ExamSession* as our next potential misbehaving entity. Only requirement *AL5* could effect the potentially misbehaving *ExamSession* activity.

Verification of Requirement AL5 with Untrusted Owner for Violation of Preconditions: The requirement *AL5* that a checker should not grade if the exam-paper is not approved by the approver is expressed below as a negation of the error situation, when eventually *Checker* starts grading without an event signifying that *Approver* has approved the exam-paper.

```
!◇ ( event(Grade_start) && !event(ApprovePaper_finish))
```

The verification failed as the *ExamSession* did not enforce the precondition for *StartExam* operation. This resulted in a violation of causality in coordination policies, and at the end, *Submit.finish* event was generated, which enabled the *Checker* to grade. To solve the violation of causality, in this case, we could have added the constraint *#(ApprovePaper.finish)=1* in the *Grader* role's admission constraints in Figure 4.11. This would ensure that participants could not join the *Grader* role as well as the *Checker* role before an *ApprovePaper.finish* event. However, we marked *ExamSession* to be managed by a trusted owner.

The second type of violation that can result from precondition manipulation is omission of events to targeted subscribers, which may cause inconsistency in coordination policies. This type of violation of global requirements due to omission of events may not be fixed by adding additional preconditions. In such cases, we require that the misbehaving entity be managed by a trusted owner.

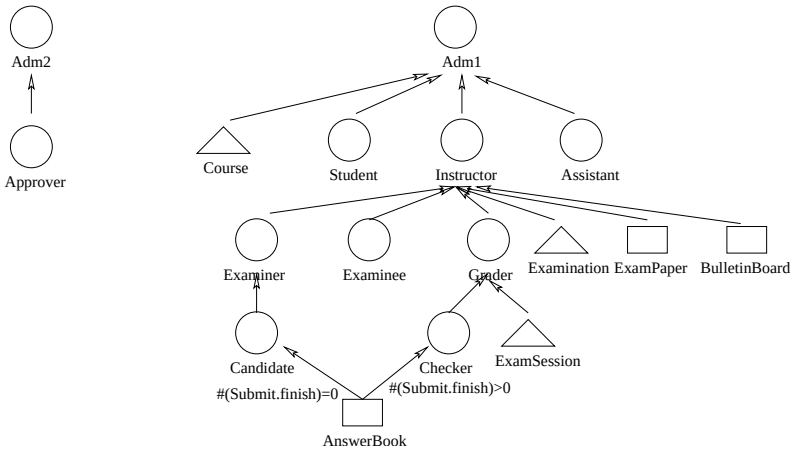

Figure 5.3. The owner assignments for the Course activity template

Case Study – Final Ownership Assignments:

As we marked all the potentially misbehaving entities as consequently misbehaving, i.e., they require to be owned by a trusted owner, we assigned *Grader* instead of *Student* as the owner of the *ExamSession* in the specification in Figure 4.12. Based on the owner assignment rules, the *Grader* becomes the owner of the nested *Checker* and *Candidate* roles. In our model, the confidentiality requirements have an implication on owner assignments. The confidentiality requirement in our example that the *Checker* role must not know participants' identities of the *Candidate* role imposes a restriction – any role whose members may be admitted to the *Checker* role, cannot be specified as the owner of the *Candidate* role. If this restriction is not imposed the confidentiality requirement is trivially violated. This requires us to assign *Examiner* as the owner of the *Candidate* roles. The resultant owner assignment for the *Examination* activity template is shown in Figure 5.3.

5.4. Verification of a Specification

Given a specification and a set of requirements, we need to ensure that the specification is correct and consistent. Among the aspects of the properties presented in the previous section, task-flow and role based constraints are correctness properties. On the other hand, confidentiality and access leakage are security properties.

As discussed in Section 5.3, we have developed, based on the aspects of global requirements to be checked, four models for efficient verification. Three of these models – *Role Model*, *Information Flow Model*, *Owner Assignment Model* – require modeling of participants for verification. These models, respectively, verify role constraints, confidentiality requirements with and without assignment of administrative roles, and integrity requirements when owners are assigned. The *Owner Assignment Model* has several modes, which takes into account the absence of trust in security domains in verifying global security requirements.

Verification of all the requirements does not guarantee that the specification is correct and consistent. By consistent specification, we mean that all the operations can be executed, i.e., all the operations are reachable with the assigned users. An activity, in our model, can be non-terminating, but there must exist an execution path for each operation. Otherwise the operation specification is redundant. To ensure correct and consistent specification, we define the term *constraint satisfaction*. *Constraint satisfaction* for an activity means that (1) all the role operations can be executed and (2) the specified constraints can fulfill all coordination and task flow constraints as well as security requirements.

5.4.1. A Bound on the Number of Participants

Our goal is to find a number, N, of users whose identities are adequate for verification of all the requirements. If a requirement is not violated for the model with N users, it will not be violated with more than N users. For a given specification, we are always concerned with a set of requirements. If there exists a bound on the number

of users for each of the requirements, then there exists a bound on the number of users for the set of the requirements.

In a verification model, the users are added to roles in non-deterministic steps. The model checker SPIN performs exhaustive search on all the possible execution paths, however, it facilitates specification to randomly select an execution path when multiple such paths exist. Using this facility, the users are assigned in non-deterministic steps, which ensures that all possible assignment-sequences of the users to the roles are searched. The goal of our procedure is to find a bound on the number of users so that none of the execution paths of a verification model is not blocked due to lack of users.

Using the following motivating examples, the main ideas behind the procedure of finding a bound on the number of users for verification are introduced.

Example 1: In the following example, there are two roles, and only a member of the *R1* role can join the *R2* role.

Role `R1` { }
Role `R2` { **AdmissionConstraints** `member(thisUser, R1)` }

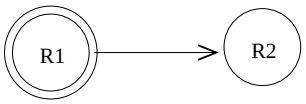

Figure 5.4. User flow graph based on Example 1

Based on the role constraints in Example 1, a *user-flow graph* is presented in Figure 5.4. In the figure, roles are presented as vertices and a directed edge from R1 to *R2* represents that a user from the role *R1* can be a member of the role *R2*. In this example, any user can join the role *R1* as *R1* does not have a prerequisite membership

constraints. The roles that do not have any prerequisite membership constraints are termed *initial assignment roles* and marked with double-circles in a *user-flow graph*. In Figure 5.4, the role *R1* is an *initial assignment role*. In this example, if we assign 1 user to the *R1* role, all the paths for users assignments to the *R1* and *R2* roles are satisfied. We represent this *user-assignment* as {R1, 1}. i.e., 1 user is assigned to R1.

Example 2: In the following example, there are two roles, and the member of *R1* role cannot join the *R2* role.

```
Role R1 { }
Role R2 { AdmissionConstraints !member(thisUser, R1) }
```

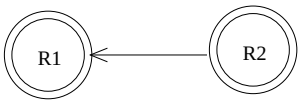

Figure 5.5. User-flow graph based on Example 2

The *user-flow graph* of Example 2 is presented in Figure 5.5. In this example, both *R1* and *R2* are *initial assignment roles*; however a user from only *R2* can join *R1*. As in the verification model, the users are added in non-deterministic steps, if 1 user is assigned to both *R1* and *R2*, all the user assignment path are executed. Hence the user-assignment in this example is { (R1, R2), 1}.

Example 3 In the following example, R1 and R2 roles cannot have a common member.

```
Role R1 { AdmissionConstraints !member(thisUser, R2) }
Role R2 { AdmissionConstraints !member(thisUser, R1) }
```

The *user-flow graph* of Example 3 is presented in Figure 5.6. In this example, both *R1* and *R2* are *initial assignment roles*; however they cannot have a common member.

Figure 5.6. User-flow graph based on Example 3

The user-assignments in this example are {R1,1} and { R2, 1}.

Example 4 In the following example, admission to the role R1 depends on the member count of the role R2. Here, R2 has to have at least N member, i.e., a user-assignment of N is required to ensure that R1 can have a member.

Role `R1` {
AdmissionConstraints `#member(R2) >= N` }

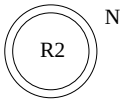

Figure 5.7. User-flow graph based on Example 4

The *user-flow graph* of Example 4 is presented in Figure 5.7. In this graph, only the role *R2* is shown and assumed that *R2* is an *initial assignment role*. Based on the required carnality constraint, R2's user count is incremented to N. If *R2* is not an *initial assignment role*, we need to ensure that *R2* can have N members after the final user-assignment.

Example 5 There can be preconditions specified to ensure that an operation can be performed if two different users have performed another operation.

Role `R1` {

```
    Operation Op1
      Precondition #Op1(invoker=thisUser)= 0 }
  Role R2 {
    Operation Op2
      Precondition #Op1= 2 }
      ......  }
```

Figure 5.8. User-count graph based on operation dependency in Example 5

To find a bound on user count based on operation dependency, we introduce the *user-count graph* in Figure 5.8. In this graph, operations are represented as nodes and a directed edge from a node *Op1* to *Op2* means that the *op2* depends on execution of the *op1*. Operations that do not depend on other operations are marked with an arrow. On a *user-count graph*, we follow the dependency path and increment user count for each operations. In this example, as shown in Figure 5.8, initially we assign 1 on *Op1* and follow the dependency graph to reach the node for *Op2*. Based on the precondition of *Op2*, we increment the count on *Op1* to be 2. In this case, *Op1* needs to be invoked by at least two users. A count assignment of 2 is marked on *Op1* in Figure 5.8.

Example 6 There can be preconditions specified to ensure that an operation Op2 can be performed if another user has performed the operation Op1.

```
  Role R1 {
    Operation Op1
      Precondition #Op2(invoker=thisUser)= 0
```

Operation `Op2`
Precondition `#Op1(invoker=thisUser)= 0 ∧ #Op1= 1}`

Figure 5.9. User-count graph based on operation dependency in Example 6

In the *user-count graph* of Example 6 is presented in Figure 5.9, we follow the dependency path and increment user count for each operation. In this example, initially we assign 1 on *Op1* and follow the dependency graph to reach the vertex for *Op2*. Based on the precondition of *Op2*, we increment the user count requirement by 1. In this case, the role *R1* needs at least two users to invoke both the operations.

Example 7 In our model, there are no constructs or mechanisms for entities in concurrent activities to reference each other. Only way entities in concurrent activities can interact is through shared objects. For concurrent activities that can have channels, we ensure that at least two such concurrent activities can be created. This ensures the execution paths where users from concurrent activities may leak information. In the following example, the operation has an action, which creates an activity and passes shared objects to nested activities.

Role `R1` {
Operation `Op1` {
Precondition `#Op1(invoker=thisUser) = 0`
Action new Activity `Activity1(obj)}`

In this example, a single object is shared among concurrent activities instantiated from the same activity template. We need to ensure that at-least 2 such concurrent

activities can be created. That is for the role R1, user-count is at-least 2. This also ensures that information can flow within a role.

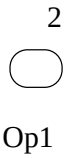

Figure 5.10. User-count graph based on object passing to a nested activity in Example 7

The *user-count graph* of Example 7 is presented in Figure 5.10. In Figure 5.10, we assign user count 2 for the operation that creates nested activities and passes objects.

The procedure to find a bound on the number of participants is facilitated by several restrictions of our specification model. These restrictions are presented as properties in the following discussion.

Property 1: *Only a fixed number of roles can have members whose memberships do not require previous memberships in other roles. In a specification, most of the role members are reflected from other roles or required to be in some prerequisite roles.*

Based on Property 1, our goal is to find a bound on the number of participants for the roles that do not have prerequisite membership constraints. We have termed these roles as *initial assignment roles* in our *user-flow graph.*

The following property is based on the scope rule of the specification model as presented in Section 4.4.3.

Property 2: *An event based condition can only refer to operations defined within*

the same activity scope.

Based on Property 2, for each activity template within a specification, there is a *user-count graph*. Each *user-count graph* can have operations as nodes from the same activity scope. Moreover, from our observation only operations that depends on invoker's identity can increment user-count in a *user-count graph*. These operations are of the form `opName(invoker = thisUser)`. Hence, if none of the operations within an activity depends on invoker identity, the *user-count graph* for that activity does not have any effect on our procedure to find a bound on the number of users.

In the following section, we present a graph-based algorithm to find a bound on the number of user-assignment for verification. For the algorithm, the following two types of graphs are used as introduced earlier with examples.

Definition: A *user-count graph* G=(V,E) is a directed graph where the vertex set V = $\{op_1, ...,op_n\}$ represents all the operations within an activity, and E is the directed edge set, where each edge $e_i = (op_i, op_j) \in E$ means that the op_i depends on execution of op_j. A subset of the vertices $V' \subseteq V$ represent operations that do not depend on execution of any other operations. We term these *initial operations*. Based on the specification model, if an activity template has operations, there is at-least one operation that does not depend on execution of any other operations. Otherwise the specification is inconsistent as it can never be executed.

Definition: A *user-flow graph* G=(V,E) is a directed graph where the vertex set V = $\{R_1, ...,R_n\}$ represents all the roles defined within an specification and E is the directed edge set, where each edge $e_i = (R_i, R_j) \in E$ represents that a member of R_i can be a member of R_j. A subset of the vertices $V' \subseteq V$ represent *initial assignment roles* and membership to these roles does not depend on users previous membership to other roles.

The *user-flow graph* can have disjoint connected subgraphs. This is due to the

fact that the specified constraints may restrict users flow from vertices of one of the subgraphs to the other. However each of the subgraphs has a subsets of the vertices that represent *initial assignment roles*. The goal of our procedure is to find a bound for the *initial assignment roles* in each of the subgraphs.

5.4.2. Algorithm to Find a Bound on the Number of Users

Following are the steps to find a bound on the number of participants for an activity template based on *user-count graph* and *user-flow graphs*.

1. [Compute *user-count graph*:] In the hierarchical activity template structure, for each activity template, compute a *user-count graph*. [2].

2. [Assign user-counts in *user-count* graphs:] Starting from the *user-count graph* of the inner most activity template,

   (a) Select an operation from the vertices in the *initial operations* set of this graph, and assign a user count of 1.

   (b) From the source operation, select the next operation based on the available edges from the source operation.

   (c) If the next operation cannot be performed with current user-count, increment the user-count of the source operation with the required user count.

   (d) If the next operation cannot be performed with the same users in the user-count, increment the user-count of the next operation with a plus (+N), where N is a user count, that is to perform the next operation, N number of new users are required.

   (e) If an operation creates an activity and passes an object in the child activity, assign a user-count of at least 2 for the operation.

   (f) Stop the procedure when all the edges are visited from all the initial operations [3].

[2] A graph is drawn only if the template has an operation that depends on invoker's identity, i.e, there is an operation of the form `opName(invoker = thisUser)`.
[3] The final user-count enables all possible operation execution paths for the activity.

(g) Repeat these steps for all the *user-count graphs* in the nested template hierarchy.

3. [Compute *user-flow graph*:] For the specification, compute the *user-flow graph*].

4. [Assign *member-count* in the *user-flow graph*:]

   (a) Based on the user-counts of operations within a role, assign a member-count for each role. This member-count is the maximum of all the user-counts within a role.

   (b) For each role, assign the largest number between the two numbers, the role's minimum cardinality and the member-count of the role found based on the *user-count graph*, as the role's initial *member-count*. If not specified in the specification, a role's default minimum member-count is 1.

   (c) If a role has maximum cardinality constraint, assign the maximum cardinality as the member-count for the role.

   (d) Repeat the previous step for all the activities.

   (e) Starting from inner-most activity template, based on the *user-flow graph*, if users can flow a role R_i of a parent activity to a role R_j in a child activity, assign the member-count of R_i as the maximum of the two member-counts. If the role R_j has a plus user-counts expressed as +N, the member-count of R_i is incremented with N.

   (f) Repeat the previous step for all the roles in the *user-flow graph*.

   (g) At the end, member-counts for all the *initial assignment roles* are assigned.

These steps of the procedure are discussed below using the example *Course* activity template in Appendix D:

1. [Compute *user-count graph*:] In the *Course* activity template, only the *Examination* activity has operations that depend on invoker's identity, i.e, there are operations of the form `opName(invoker = thisUser)`. The *user-count graph* for the operations of this activity is presented in Figure 5.11.

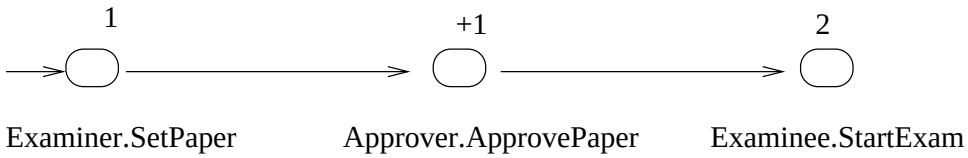

Figure 5.11. User-count graph based on operation in *Examination* activity

2. [Assign user-counts in *user-count graphs*:]

   (a) In Figure 5.11, *Examiner.SetPaper* is the only operation in the *initial operations* set. A user count of 1 for the *Examiner.SetPaper* operation is assigned.

   (b) The only next operation is *Approver.ApprovePaper* based on the available edges from the operation *Examiner.SetPaper*. As the *Approver.ApprovePaper* operation cannot be performed with the same user, increment user-count with a plus one, i.e., +1, for the operation *Approver.ApprovePaper*.

   (c) Based on the edges representing operation dependency, the next operation is *Examinee.StartExam*. This operation does not depend on the users who has invoked the other operations. However, this operation creates an activity and passes an object. A user-count of 2 is assigned for the operation *Examinee.StartExam*.

   (d) After *Examinee.StartExam* the procedure to assign user-count stops, as all the edges of the graph are visited.

3. [Compute *user-flow graph*:] The *user-flow graph* for the specification is presented in Figure 5.12. In this *user-flow graph* initial assignment roles are *Assistant*, *Instructor*, *Student*, and *Approver*.

4. [Assign member-count in the *user-flow graph*:]

   (a) The *Candidate* and *Checker* roles in the inner most *ExamSession* activity template have a maximum cardinality constraint of `1`. A member-count of

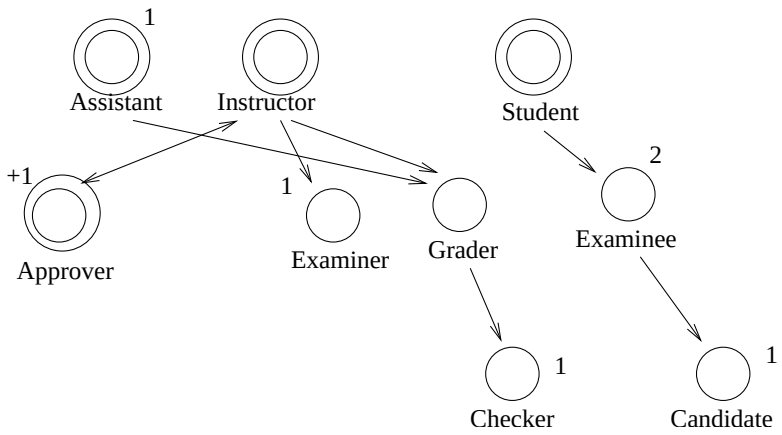

Figure 5.12. User-flow graph of the *Course* activity

`1` is assigned to each of the roles for the inner most *ExamSession* activity template in Figure 5.12.

(b) For the parent activity *Examination*, based on the *user-count graph* in Figure 5.11, the member-count for the *Examiner*, *Approver*, and *Examinee* are updated.

(c) At the end, the member-counts for the *initial assignment roles* are 1 for *Assistant*, 2 for *Student*, 1 for *Approver*, and 2 for *Instructor*. The *Instructor* role requires 2 members based on the member-count of the *Approver* and *Examiner* roles, as expressed by *+1* and *1* in the graph. However, *Approver* and *Instructor* roles can share members, and we assign the same 2 members for these two roles.

5. Hence the final member-count for the *Course* activity template based on our algorithm is `5` with `2` in *Student*, `1` in *Assistant*, and shared `2` in *Instructor* and *Approver* roles.

5.4.3. Correctness of the Algorithm

We rephrase the goal of the the algorithm presented in the previous section to prove its correctness. The algorithm finds a bounded number of users for the set of initial

assignment roles, such that the verification run does not miss a possible execution trial which may violate a property. That is the search space contains execution trials, which are equivalent to all possible execution trials of policy violations.

Property 3: *Parts of the specification model that require an invoker identity are specified as: (1)* `member(thisUser, roleId)`*, and (2)* `opName(invoker = thisUser)`.

To prove the correctness of the algorithm, we rely on Property 3 that none of the requirements is specified based on participants' identities but rather on participants' role memberships and their previously performed operations. We define the term *invoker credential* representing the conditions that an invoker has to satisfy to invoke a specific operation.

Definition 3: *Invoker credential* represents the role memberships and the operation precondition that an invoker has to satisfy to invoke a specific operation. If requirements related to information flow are specified, the *Invoker credentials* also represent the shared objects that can be accessed by invoking the operation.

An operation can have multiple *invoker credentials* as the invoker can acquire a role membership based on memberships to different set of roles. Moreover, an operation precondition can have disjunctive conditions resulting in more than one *invoker-credentials* for the operation. The idea behind the algorithm is that if two users *u1* and *u2* with same *invoker-credentials* can violate a property, a verification run with either one of them is sufficient. The goal of our algorithm is to find a minimal number of participants that can generate all possible *invoker-credentials* when assigned to roles in non-deterministic steps.

Now, we discuss the restrictions, specified below as properties in the specification model, that ensure a bound on the participant count for a given requirement.

Property 4: *The specification can have a constraint specifying the number of required members for a role.*

For example, if a requirement specifies that a role must have `N` members, our verification model requires at least `N` participants.

Property 5: *A role membership related condition can only refer to roles defined in the scope of ancestor or current activities.*

The nested structure of the activity template creates an activity instance tree with each instance as a root for all its ancestor activity instances. As there is always a fixed number of roles in ancestor activities as well as within an activity instance, for the role model, there is a bound on the number of *invoker credentials*.

Property 6: *An event based condition can only refer to operations defined within the same activity scope.*

Due to this property, we can enumerate all *invoker credentials* for a role operation within an activity.

Property 7: Information can only flow through storage channels that are shared objects or information objects.

As the *invoker credentials* also takes into account shared objects, a user assignment have all possible channels for a given requirement.

Based on the properties 4, 5, 6, and 7, for a given requirement there is a bounded number of *invoker credentials* for all the role operations. However, more than one assignments of users in *initial assignment roles* may be required to ensure execution paths in the model checker covers the *invoker credentials* for a requirement. As participants are assigned to roles in non-deterministic steps, this assignment ensures all

possible *invoker credentials* for the role model. If the minimal number for the requirement is less than the required number as expressed by property 4, the model checker either reports an unreachable state showing that an operation cannot be executed or provides a trace showing that a larger number of user is required to satisfy the requirement.

Now we need to prove that the user-assignment based on these graphs generates all possible *invoker credentials*. In this thesis, the semantics of our specification model is not presented based on the semantics of the PROMELA specification model. In our future work, we can formally prove that the algorithm finds the user count that generates all the possible *invoker credentials*, given a specification. Here we provide the proof sketch that there exists a user assignment in roles that satisfies role membership related constrains, which results in a *user-flow graph*, and operation preconditions, which results in *user-count graphs*.

In the role verification model that is concerned with only role membership related constraints, to generate all possible *invoker credentials* the user-assignment in the *user-flow graph* has four distinct scenarios:

1. A single partition of the *user-flow graph* with assignment of a single user in all the roles that ensures all the possible *invoker credentials*. This scenario is presented in Example 1 and Example 2. In Example 1, assignment of a single user in the *initial assignment role* R1, i.e., {R1, 1} ensures all the possible *invoker credentials*. These are for R1 is {null} and for R2 is {R1}.

2. The *user-flow graph* is partitioned into distinct sub-graphs due to constraints that require disjoint members. This scenario is presented in Example 3.

3. The member count of a sub-graph has to be incremented to satisfy specific constraints as shown with the Example 4. In our algorithm for minimal count, we mark these sub-graphs with the required member count.

4. Lastly, the specified constraints cannot be satisfied and some roles cannot have members due to inconsistent specification. With any member-count, the verification run returns unreachable states, i.e., roles that cannot have members, for this scenario.

In the owner assignment or access leakage model, to generates all possible *invoker credentials* for a specific operation, we need to generate all possible *invoker credentials* that are specified in the preconditions for the operations. Lastly, in the information flow model, when sharing of object may results in information channels, two users with same *invoker credentials* need to be present to ensure information flow among users with same *invoker credentials*.

The ideas that the assigned member-count based on the algorithm generates all the possible *invoker credentials* are discussed below using our example *Course* activity template.

1. The *Candidate* and *Checker* roles in the inner most *ExamSession* activity template have a maximum cardinality constraint of `1`. The algorithm needs to ensure that users with all the possible *invoker credentials* can join these roles. Based on Figure 5.12, the *invoker credentials* for the *Checker* role are { *Assistant*, *Grader*}, { *Instructor*, *Grader*}, and { *Approver*, *Instructor*, *Grader*}. As the member-count is assigned from the inner-most activity, the algorithm ensures that all the possible *invoker credentials* are preserved by assigning member-count on the roles in the parent activity. This process of member-count assignment is recursive thus ensuring all the possible *invoker credentials* for the inner roles.

2. In our example, the *ApprovePaper* and the *StartExam* operations have invoker identity dependent "intra-role coordination" requirements. The *invoker credentials* for the *ApprovePaper* operation are:

   {*Adm2*, *#(SetPaper.finish(invoker=thisUser))=0*} and {*Instructor*, *#(SetPaper.finish(invoker=thisUser))=0*}.

These *invoker credentials* are derived by adding the operation precondition of the *ApprovePaper* operation based on the *invoker credentials* in the role verification mode, i.e.,

$\{Adm2\}$ and $\{Instructor\}$.

Addition of the operation precondition in *invoker credential* does not effect the number of *invoker credentials* for this operation as there is only one operation for this role. Similarly, the *StartExam* operation's *invoker credential* count remains same with the addition of the operation preconditions.

3. Due to the shared objects – *BulletinBoard* and *ExamPaper* – concurrent *Examination* and *ExamSession* activity instances share information channels. However, it is specified that only one instance of the *Examination* activity can be created. For probable channels among multiple instances of *ExamSession* activities, at least two of the *ExamSession* instances need to be created. For the minimal count, this requires at least two members for the *Examinee* role, which in turn requires two members for the *Student* role.

Lastly, we need to shown that for a specific requirement, the number of *invoker credentials* for the derived minimal number, remains same for any larger number of participants. Given that the derived minimal number is N for a specific requirement of an activity template T. For K (K>N) number of users participate in an activity instance, assume that the set of *invoker credentials* is C. We need to show that the set C remains same when K+1 users participate in an instance of T. As the algorithm to find a minimal number only depends on the specification, the increment of participant count does not have any effect on the minimal number. Based on induction, as the *invoker credentials* set C remains same for any number of participants larger than N. As there exist a minimal number of participants for each specific types of requirement, there exist a minimal number of participants for a set of requirements. Conversely, If there exist minimal number for all the requirements than there exists, then the number is also minimal for each of the requirements.

5.5. Related Work: Model Checking Security Properties

Background discussion on security models and formal verification of such models are presented in Chapter 3. Researchers, e.g., (Focardi and Gorrieri, 1997; Schneider, 1996; Volpano and Smith, 2000; Fine, 1996; Simonet, 2002; Martinelli, 1998) have utilized formal methods for verification of security properties, mainly of various types of confidentiality properties (Zakinthinos and Lee, 1997). However, these approaches assume that the verified programs will run under trusted subjects. In decentralized administration of systems, programs are type-checked based on assigned "trust" (Ørbæk and Palsberg, 1997) or labeling data (Myers and Liskov, 2000); however, the execution environment is assumed to be entirely trusted. In this thesis, we have utilized finite state based model checking to verify security properties in collaboration systems. In our verification approach, some of the distributed security domains can be under the control of administrators who may not be trusted, and the behavior of untrusted administrators are modeled to verify security properties.

Compared to other research in finite state based model checking, such as verification of workflow processes (Eshuis and Wieringa, 2002; Janssen et al., 1998), program analysis (Corbett et al., 2000) and protocol verification (Maggi and Sisto, 2002), our verification of role based security specifications needs a bound on the number of participants for a verification to tackle state space explosion. In this thesis, we provide based on case studies, a framework to find such a bound on the number of participants.

5.6. Summary

In this chapter, we have presented a verification methodology for a collaboration specification. First, different types of coordination and security properties that need to be verified are discussed and classified. Second, finite state based model checking is utilized for the verification of the properties. And lastly, challenges in model checking security policies in collaboration systems are discussed, and solutions are presented.

Chapter 6

Design of a Policy-Driven Secure Middleware

In the previous chapters, we have discussed a specification model and a methodology to specify collaboration environments. In this chapter, the design of a middleware that is implemented as a proof of concept for constructing collaboration environments is presented. In our model, a policy-driven collaboration system is realized in three steps as shown in Figure 6.1.

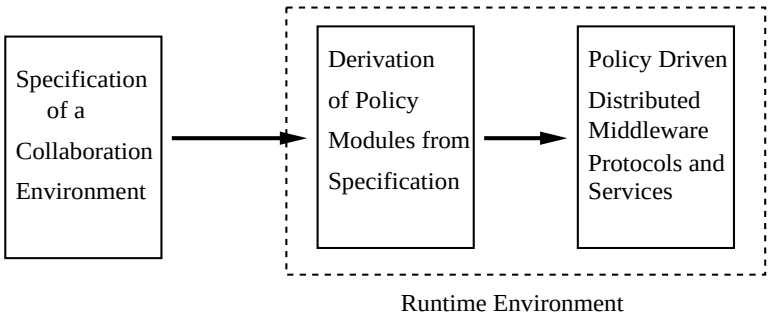

Figure 6.1. Overview of the middleware for distributed collaboration systems

The primary function of the middleware is to execute a collaboration environment

based on its specification at distributed hosts that may reside in different security domains. This requires that the middleware provides protected execution environments in distributed nodes for enforcement of policies. Our motivation is to enforce only coordination and security policies. Based on the specification model, the policy aspects related to security, coordination, and meta-administrative policies are generalized under three entities – activity, role, and shared object – for proper enforcement of a collaboration specification. Based on our administrative model, the owners of these entities are trusted for enforcement of specific policy aspects of a collaboration specification.

In the next section, we present the design challenges of our policy driven middleware. The execution environment of our middleware including middleware components and the runtime data structure are presented in Section 6.2. In Section 6.3 and Section 6.4, the design of middleware services and the functional design of the middleware components are discussed, respectively.

6.1. Design Challenges of Policy-Driven Middleware

The distributed architecture and security considerations of the policy-driven middleware impose several design challenges. For runtime environment of the middleware, the middleware needs to support selection of secure sites that can be trusted for policy enforcement. Event is the building block for maintaining consistent views among the distributed components of the system. Secure event service for integrity, confidentiality, and availability of events is a primary design concern. Primary task of a collaboration system is to coordinate the collaborating users. Maintaining consistent coordination states is a challenge for distributed entities of the middleware. Moreover, the middleware has to facilitate sessions for users' interactions with other users and shared objects and resources. Lastly, for a policy driven middleware, design of the middleware needs to be flexible to adapt with the changes of the policy specification model. These requirements are discussed in the following subsections.

6.1.1. Secure Sites for Policy Enforcement and Object Storage

If the site where an activity is managed is not secure, trust on the activity owner with the responsibility for that activity's management cannot ensure proper enforcement of policies. Similar challenges exist in maintaining secure sites for role, activity template, and object management. For example, only in secure sites of a role owner, enforcement of the role's membership information and role membership operations can be secure. Arbitrary storage location of an object cannot be trusted for proper enforcement of object access control policies. Ownership and other access control policy of an object can change in different stages of a workflow. If a trusted site is selected during object creation by the object creator, that site may not be trusted at a later stage of the workflow.

Not being able to trust any single site for all management tasks imposes constraints on the middleware in supporting the security requirements of access control, authentication, confidentiality, and non-repudiation. Mechanisms are required to find appropriate nodes, which can be trusted to enforce various security policies derived from the collaboration specification. Our policy specification model and the corresponding methodology assign trust to an owner role for management of specific entity. However, assignment of secure sites for an owner is performed at runtime, that in turn requires secure execution environments in those sites.

6.1.2. Policy Module Creation and Distribution

Initially, when an activity instance in realized from an activity template, different types of security policies are required to manage a collaboration environment. These security policies, termed policy modules, are based on roles and object types in the context of activity instances that will be created at runtime. For example, for each role in an activity instance, policy modules for admission, activation, and validation constraints are required. Policy modules for role operations require preconditions together with the directives for subscribing events from other entities in the system for evaluating these preconditions. For objects, the role-based access control policy modules identify the method invocation privileges that should be given to a role in a

specific activity instance, together with the role operation context in which the methods can be invoked. Similarly for an activity template, a policy module is required that defines the activity creation privileges for authorized roles.

An activity instance can have nested templates for other activities that are instantiated at runtime. During instantiation of the parent activity instance, this parent activity context information needs to be securely maintained for proper generation of policy modules for the nested activity instances. Lastly, in a distributed environment, these policy modules may need to be enforced in different sites. Secure distribution of policy modules is a challenge in the design of the policy based middleware.

6.1.3. Distributed Trust Relationships

Users participate in a collaboration through roles, and role membership certificates are generated at distributed sites. Policies are required that specify who can create and manage role certificates. Role membership certificate may not be adequate for policy enforcement as participants may need to be uniquely identified for inter-role coordination, i.e, the participant may need to be uniquely identified in the context of a role. In some cases, participants are admitted to a role based on their membership in other roles. If membership certificates are used for enforcing such admission conditions, a trust relationship needs to be maintained among participating sites to enforce the authenticity of such certificates. Lastly, a user may require an identity certificate to be admitted in the system. Similarly, unique identity certificates may be required for shared objects and resources to enforce access control.

6.1.4. Secure Distributed Event Service

In our model, coordination and security policies are specified and enforced based on various kinds of events. Therefore, security of event communication requires that events are not tampered with, are generated by correct notifiers, and only received by authorized subscribers. Importantly, when roles or shared objects are maintained in different trust domains, the middleware needs to ensure that events are not *falsified* or *omitted*, i.e., the notification of the events can be trusted. Moreover, collaborative

applications have several unique access control requirements on the event data as discussed below.

- Event Confidentiality: The information flow through the events needs to be secured. Moreover, information should not flow to unauthorized users. Certain information or attribute of an event may need to be deleted depending on the receivers of the event according to access control policy. Event data may need to carry pseudonyms instead of actual identity or location of events' publishers.

- Fine Grain Access Control: The granularity of access control on the event can be fine grained. For example, the identity of a reviewer may need to be secured when he publishes his reviews to other reviewers. If such an activity generates an event, the identity attribute of the event publisher needs to be hidden from other reviewers.

- Event Causality: The presence of events in a collaboration provides strong casual and temporal relations among collaborating users. They may also leak information about the collaboration protocol and other collaboration specific activities. Hence the information of the occurrence of an event may need to be treated as secure data. Similar requirements on securing event causality, specifically causal denial and forgery are addressed in (Reiter and Gong, 1995).

The flow and integrity of information are well studied in multiuser systems. However, these systems provide a rigid requirement on classifying data and are usually realized through centralized databases. Such models are too restrictive for collaborating users for their real-time requirements, as the events have to pass through the central database. For realtime events, the database introduces a bottle-neck for large number of collaborations. Most existing solutions on the security of events concentrate mainly on the integrity of events and propose various transport or session layer

encryption mechanisms like TSL or SSH. In some cases, the events are treated as persistent data in databases, and conventional database authorization mechanisms are proposed to enforce access control.

6.1.5. Coordination Consistency Management

In decentralized management of a collaboration, correct enforcement of coordination constraints is an important problem as the system state is distributed and a node's view of it may be incomplete or stale. As dynamic access control policies are closely tied to coordination constraints, security of the system requires consistency in evaluation of such constraints.

6.1.6. Secure Session Management

The middleware needs to provide support for users to join or leave a collaboration. Similarly, the owners should be able to admit or remove users from a collaboration. A user may not be present or active when specific role memberships are assigned to him/her. Moreover, the user may try to join in later stages of a collaboration activity. These situations require both *push* and *pull* models of user assignments to be supported by the middleware. In the push model, users are admitted to a collaboration by an initiator of a collaboration activity and users' membership assignments are pushed to the users. In the pull model, a user can pull all the collaboration activities where he or she can join to participate.

A session may manipulate shared objects. For each method invocation, the object server needs to verify the invoker's id, role, and the name of the role operation in which this method is being invoked. It should also verify the operation precondition and access rights of the invoker. When an object session is initiated it can be either short lived with a predefined order of object accesses or long lived with user accessing the shared objects based on his/her access-rights. Middleware needs to support mechanisms for secure management of both short and long lived sessions.

6.1.7. Adaptation with the Changes in the Specification Model

The primary function of our policy based middleware is to enforce policies derived from a collaboration specification. The specification model is only concerned with coordination and security policies. Moreover, the middleware treats the specification model as an integration specification of shared applications and objects. As collaboration systems are emerging, we may require new constructs to express emerging new kinds of policies for security and coordination. It is desirable to incorporate in the design of the policy based middleware the provision to adapt with the changes in the specification model.

6.2. Architecture of the Policy-Driven Middleware

There are three independent utilities that are required to execute a collaboration environment. These are *Trusted Server*, *User Coordination Interface (UCI)*, and *NameRegistry*.

- `Trusted Server:` *Trusted server* is the primary middleware component executed at distributed hosts. The middleware supports many different collaborative activities, possibly unrelated to each other and created by different participants. The primary function of a trusted server is to provide support for protection domains for owners or administrative roles. Along with the security service of the middleware, a trusted server forms the TCB (*trusted computing base*) for collaboration environments.

- `UCI:` The system provides each user with a coordinator on his/her system, the *User Coordination Interface (UCI)*, which maintains the user's view of collaboration activities in different trusted servers and provides suitable interfaces to the user.

- `Name Registry:` The *name registry* is responsible for managing names. Management of names includes (1) providing name certificates and (2) enforcing namespace ownership enabling namespace delegation. Any decentralized name management service that provides the above two services can be utilized for

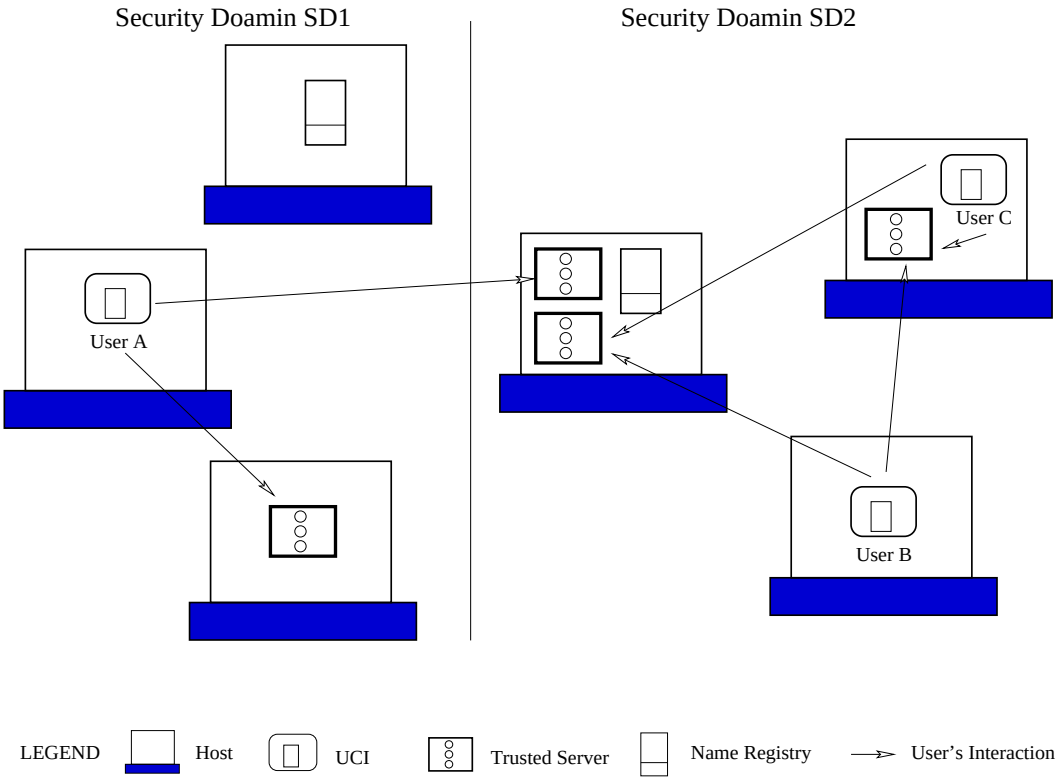

Figure 6.2. Application components in distributed collaboration environment

our collaboration middleware. We use the Ajanta name service (Tripathi et al., 2002) for this purpose.

In Figure 6.2, the execution environment, composed of the application level components of the middleware, is presented. Each host can run trusted servers, UCIs, or a name registry. The UCI for a trusted server need not to be on the same host. On the other hand, a user with his/her UCI can interact with different activities managed in multiple trusted servers. Each administrative domain usually manages a name registry as shown in the Figure 6.2.

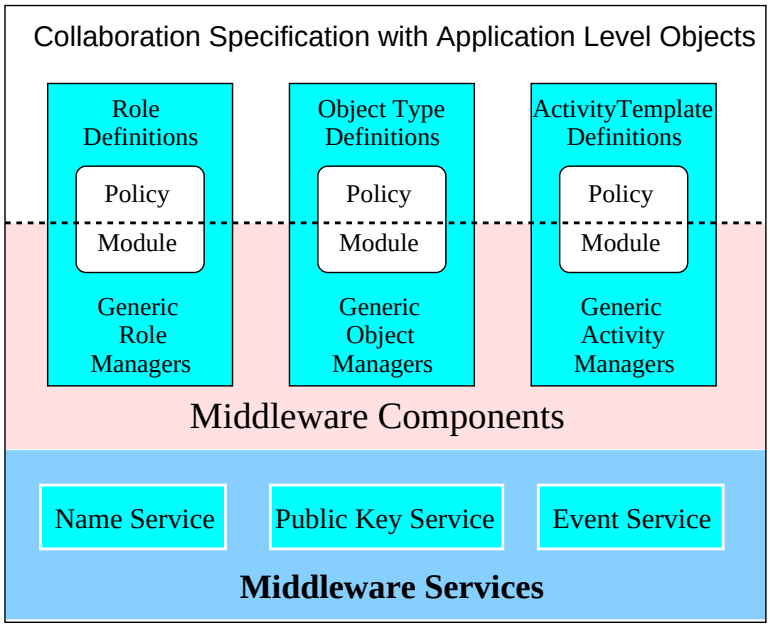

Figure 6.3. Middleware components and services

Figure 6.3 presents the components and services for our policy-driven middleware at three different levels. Based on the specification model, definitions for roles, object types, and activity-templates are manipulated at the application level. Based on these definitions, policy modules are generated for management aspects of roles, activities, and objects. These components and services are primarily managed at trusted servers running in different administrative domains. Trusted servers interact with the name registries to provide name service, public key certificates for the users and other entities in its environment.

In Figure 6.3, three managers – *role*, *object type*, and *activity template* – are presented. A *manager* is responsible to enforce the entity specific policies at runtime. Based on a collaboration specification, policy templates or modules for management of these entities are generated by the middleware. However, two other managers – *object manager* and *activity manager* – are supported by the middleware to manage the runtime instances of object types and activity instances. An instance of the manager

is created based on a generic manager with the entity specific policy module.

6.2.1. Runtime Data Structure

Every manger in our model is represented as a tree structure based on their names in the nested activity template structure. This tree structure representation is used both for system-level management of an activity as well as for building user-level interfaces for the participants. A manager in the tree of an activity represents a nested entity such as an activity template, an activity instance, an object type, an object instance, or a role. A manager representing a nested activity forms a subtree within its parent activity's tree. The primary function of a manager is to maintain data structure for runtime information needed for enforcing its policies. These managers are utilized with corresponding *remote interfaces* to provide access to management functions for the entity that it represents. For example,

- The manager for an activity template supports functions for creating new instances in compliance with the security policies, and keeping track of all instances of the activity template. For each activity instance manager, it maintains an RMI reference.
- A manager corresponding to a role is responsible for admitting/removing users from the role and enforcing admission constraints when a user joins a role. It also provides to the participants interfaces for invoking role-specific operations subject to coordination conditions.
- Object type managers have functions similar to the activity template managers, they decide which roles can create an instance of that type, and maintain references to all the object instance managers that it has created. Object instance managers are responsible for object storage and authenticated invocation of its methods.

The manager tree graph of a collaboration environment changes at runtime as new activities or objects are created.

In a distributed environment this tree structure is utilized to provide each participant a view of the collaboration environment based on his/her role-based privileges. We refer to this as a *role-specific view*. A manager representing an entity in the tree can be visible to multiple participants. The manager of a specific entity is maintained at a trusted server and is designated as the *primary node* for that entity. The manager is responsible for managing all security sensitive functions and event subscriptions for the entity it represents. All other users which see that entity have a *RMI stub* to communicate with the manager. The stub contains the protocols for authenticated invocation of functions at the manager. The managers are maintained in trusted servers and the stubs are distributed to specific users' *UCIs*.

Whenever a user joins the environment, he/she first sees only a default root-level activity, termed *System*. To realize a new collaboration system, a user in the *Convener* role in the *System* activity select an activity template and installs it under the *System* node. A user can list all the activities that have been defined under *System* and that he/she can join. The manager for the activity is located at the owner's *Trusted Server*.

Figures 6.4, 6.5, 6.6, and 6.7 illustrate the execution model for our example *Course* activity template. These figures show the role-specific views for a convener X and users A, B, and C. This illustration uses eleven kinds of nodes in the object graphs at different users' sites. There are two kinds of nodes — manager and stub – for each of the five mangers: activity template, activity instance, object type, object instance, and role. For a role stub, we use shading to indicate if a participant is a member in that role.

In Figure 6.4, user X being the convener has loaded activity templates for *Course* and *Meeting*. The corresponding managers are created in a trusted server in a protection domain under the *Convener* ownership. The other users can query existing *system* activities initiated by specific user. In Figure 6.5, the convener has created an instance of the *Course* activity, *chemistry*. By default, the *Convener* is the owner

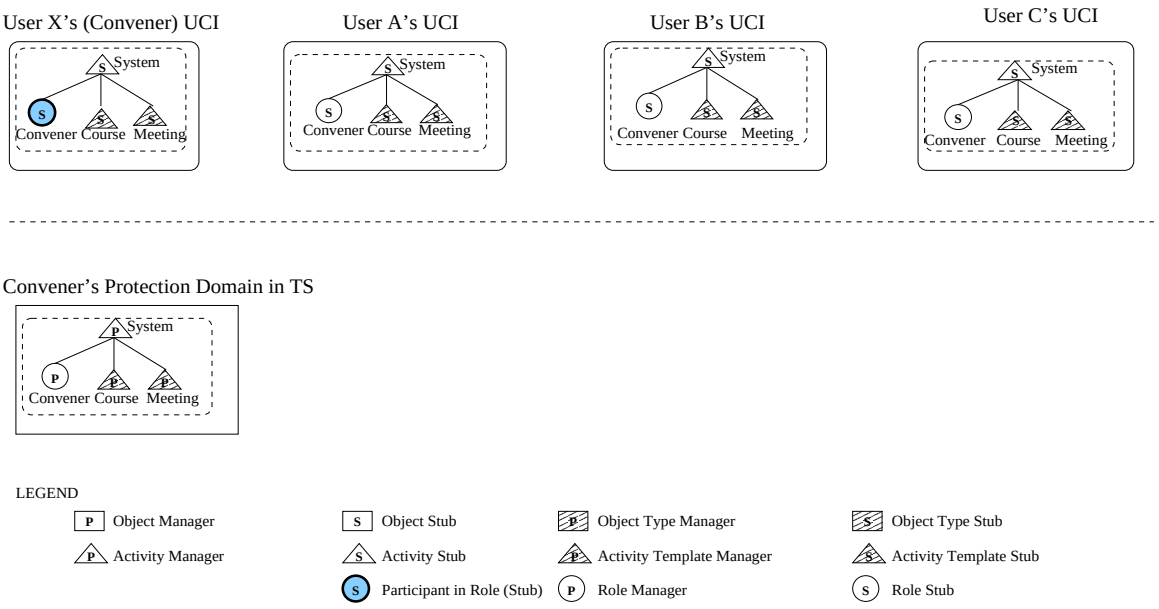

Figure 6.4. System level view in TS and UCI: After User X being a Convener instantiated a Course activity. Before users A, B, C join roles in this activity

of the *chemistry course* activity, and the manager of this activity reside at the *Convener's* protection domain. Assuming that user A is assigned to the *Instructor* role, user B to the *Assistant* role, and user C to the *Student* role, the role specific views of the *chemistry* course activity are distributed to the corresponding users' UCIs as shown in Figure 6.5.

According to the specification, the *Instructor* can instantiate an *Examination* activity through its stub and is the owner of that instance. In Figure 6.6, user A in the *Instructor* role has created an instance of the *Examination* activity *midterm exam*. Users A, B, and C are assigned to roles in this activity by role reflection as specified in the *Examination* activity template. Their views get updated by the owner of the activity. User A is now a participant of the *Instructor*, the *Examiner*, and the *Grader* roles. Being the owner of the *midterm exam* activity, *Instructor* role's protection

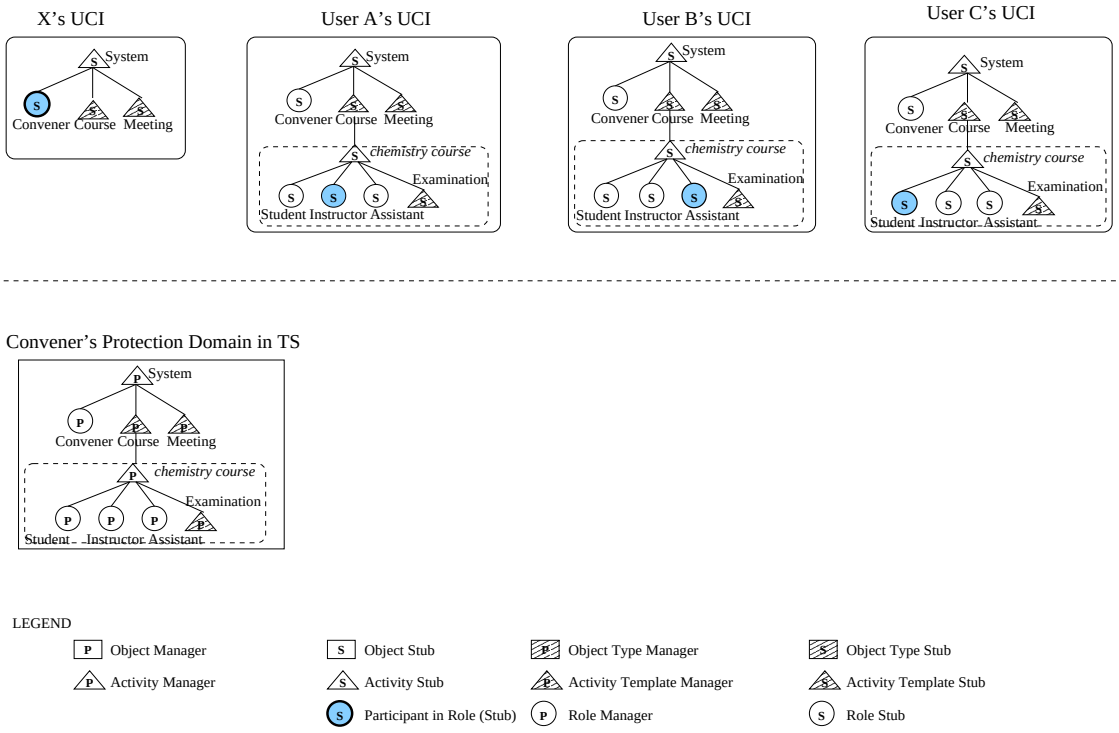

Figure 6.5. Continuation from Figure 6.4: After joining Course activity, users A, B, C get their role specific views

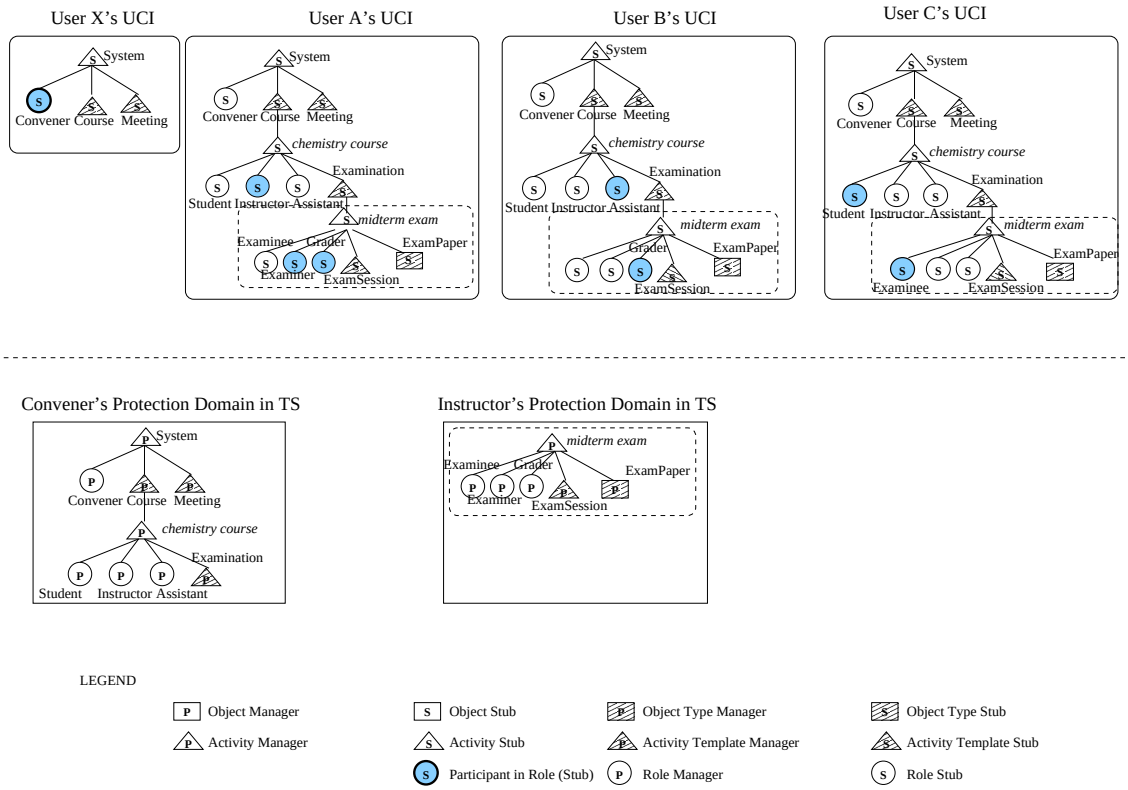

Figure 6.6. Continuation from Figure 6.5: After user A has initiated an Examination Activity

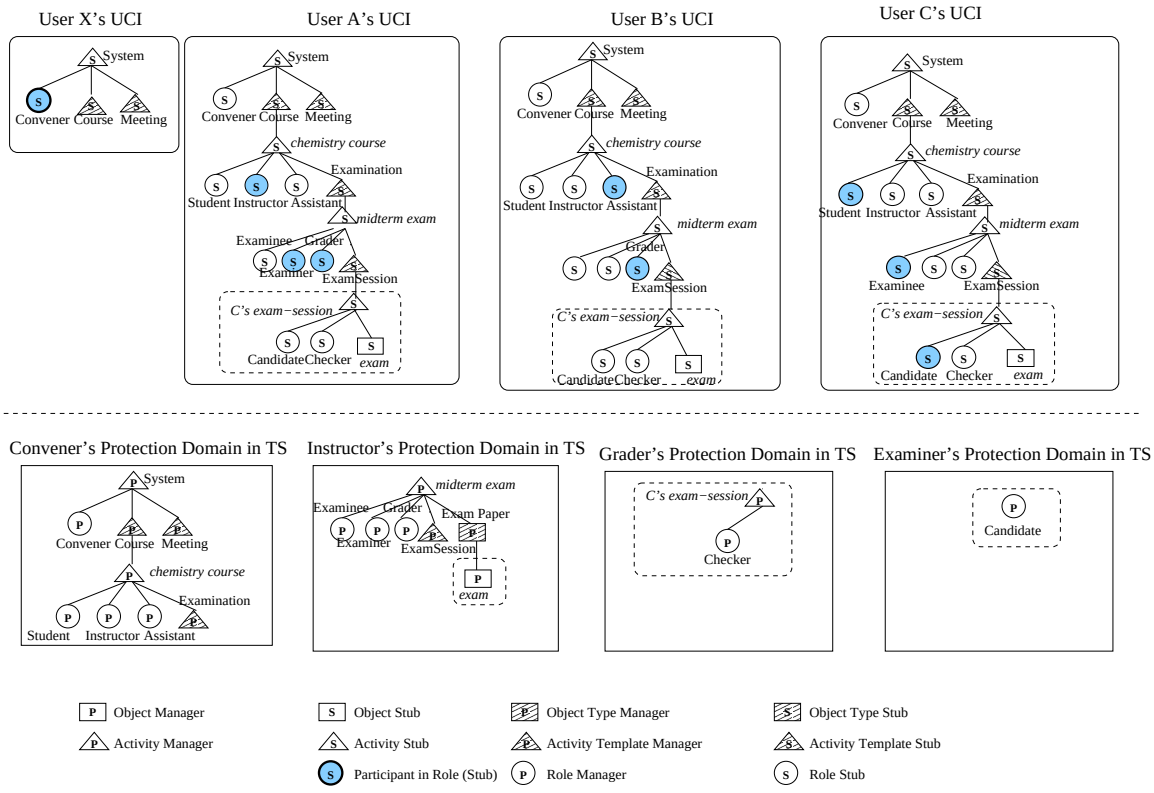

Figure 6.7. Continuation from Figure 6.6: After user C has initiated an ExamSession Activity

domain maintains the managers of this activity's roles, and nested managers for *ExamSession* template and *ExamPaper* object type. If the participants of the *Instructor* role, in this case only A, do not specify any preference and policies for selecting a trusted server for its protection domain, by default rule the trusted server of the owner of the *Instructor* role, i.e., the *Convener's* trusted server, will be selected for the *Instructor's* protection domain. In this case, entities owned by both the *Convener* and the *Instructor* roles will be in the same trusted server but in different protection domains.

In Figure 6.7, the *Examiner* creates an *ExamPaper* object *exam*, and user C instantiates an *ExamSession* activity. Based on the owner assignments, these managers are created on the corresponding protection domains.

In Figure 6.7, the managers are shown in protection domains of the corresponding owner roles. These managers can reside in the same trusted server or in different trusted servers based on the trusted server selection policies of the owner roles.

6.2.2. Policy Module Derivation

Every manager maintains various policy modules for access control and event subscription and notification. These policy modules are initially derived from an activity template and are modified when instances of the activity are created. Moreover, policy modules are specified as *policy templates* in the context of activity instances, i.e, a policy is specified based on activity types, roles, and object types. When an activity template is instantiated, the first step is to derive policy modules for the following management aspects.

- *Event Subscription and Notification Policy :* During activity creation, the activity manager derives the *notification policy* and the *subscription policy* modules for every role within its scope. The *notification policy* for a role contains information about the other roles that would be subscribing events from it, while the *subscription policy* lists the events that this role expects to receive from other entities; these are the events that influence the admission and activation

```
OBJECT=ACTIVITY(Course, $x).ACTIVITY(Examination, $y).OBJECT(ExamPaper, $z)
OWNER=$x.Instructor
ENTRY{
  SUBJECT=$y.Examiner
  PERMISSION=setPaper
    CONDITION=#($y.Examiner.SetPaper.start)=0)}
ENTRY{
  SUBJECT {ACTIVITY_TEMPLATE=$y.ExamSession, ROLE=Candidate)
  PERMISSION=readPaper
    CONDITION=null }
```

Figure 6.8. Template based access control policy module for the ExamPaper object

constraints as well as the preconditions of the role operations. All incoming events are checked against this policy to ensure their integrity.

- *Object and Activity Instance Creation Policy: Object creation policy* specifies the condition under which members of a role can create a specific number of instances of an object type. Similarly, *activity instance creation policy* specifies the condition under which members of a role can create a specific number of instances of an activity template.

- *Object Access Control Policy :* This policy module specifies the conditions under which a user can access methods of a specific object instance.

The access control policy template for the *ExamPaper* object in our *Course* activity template is shown in Figure 6.8. The *ExamPaper* object instance z is specified in a nested context of a *Course* x and an *Examination* y activity. Object z is owned by the *Instructor* role in *Course* x activity. The first entry specifies that the *Examiner* in the *Examination* y instance can invoke the *setPaper* method on object z when the specified conditions are satisfied. This event condition is subscribed from the manager of the *Examiner* role of the *Examination* y instance. In the second access control entry, the policy for invoking *readPaper* method on object z can only be realized when a

```
OBJECT=Course.chemistry.Examination.midterm.ExamPaper.exam
OWNER=Course.chemistry.Instructor

ENTRY {
 SUBJECT=Course.chemistry.Examination.midterm.Examiner
 PERMISSION=setPaper
  CONDITION=#(Course.chemistry.Examination.midterm.Examiner.SetPaper.start)=0)}
ENTRY{
 SUBJECT {ACTIVITY_TEMPLATE=Course.chemistry.Examination.midterm.ExamSession,
          ROLE: Candidate}
 PERMISSION: readPaper
  CONDITION: null}
```

Figure 6.9. Binding of access control policy at runtime of policy module in Figure 6.8

Candidate role is created in an instance of the *ExamSession* activity template. This privilege is specified using the context of an activity template and a role. As new instances of activities, roles, or objects are created, binding takes place in the above policy template. The resultant policy templates are distributed to the manager nodes of the newly created entities. As activities and objects get instantiated at every stage in Figure 6.7, the above template becomes more specific to the object, and finally fully instantiated when the *exam* object is created. The final access control policy module for the *exam* object is shown in Figure 6.9.

Similarly, other policy template modules are installed with the manager managing the corresponding entities.

6.2.3. Design of a Trusted Server

The specification model presented in the previous section is expressed using an XML schema, which represents the syntax as well as the semantics of an activity template. Based on the XML schema, different activity template specifications, such as – *Course*, *CollboraiveAuthoring*, *OnlineMeetings* – are developed through an *Editor*

application.

At runtime, activity instance specific information needs to be maintained, For example names and identities of each activity instances, values of predicates on operation preconditions or other constraints. These values are strongly tied with the scope of the activity template where the current activity instance is created. For example, in the above example, the runtime names and values maintained for the entities in the *midterm* examination activity instance of the *chemistry* course is different from other courses that may have been instantiated from the *Course* activity template. To maintain this instance specific information, an additional XML schema for runtime data is defined. The runtime information is kept based on this schema in a DOM tree as shown in the Figure 6.10. Each manager is provided with a reference to the corresponding node in the XML DOM tree.

In the trusted server, the runtime information of activity instances is maintained in an XML DOM structure based on the XML schema. The runtime DOM tree structure provides similar functionalities of an attribute tree of a compiler: it maintains value and predicate evaluation rules for each expressions and scope for each entities for proper type enforcement. For example, in Figure 6.9, an *ExamPaper* object created in the scope of the *chemistry* activity has to enforce the policy module that is different from any other *Course* activity instances.

In Figure 6.10, basic components of a trusted server are presented. The trusted server utilizes the *Object Migration* and *Public Key Certificate Management* service of the Ajanta mobile agent platform. The other components of a trusted server are *Manager Table*, *Runtime Data Structure in XML DOM*, and *Event Dispatcher*. The *Trusted Server* interface provides remote APIs for creating managers by passing manager specific policy modules. Using *createManager* APIs, managers are added in the *ManagerTable* component of the middleware. *createManager* calls from the same trusted server do not use the RMI interface. A function of the manager table is to maintain protection domains for specific owners. When an *Activity* manager is

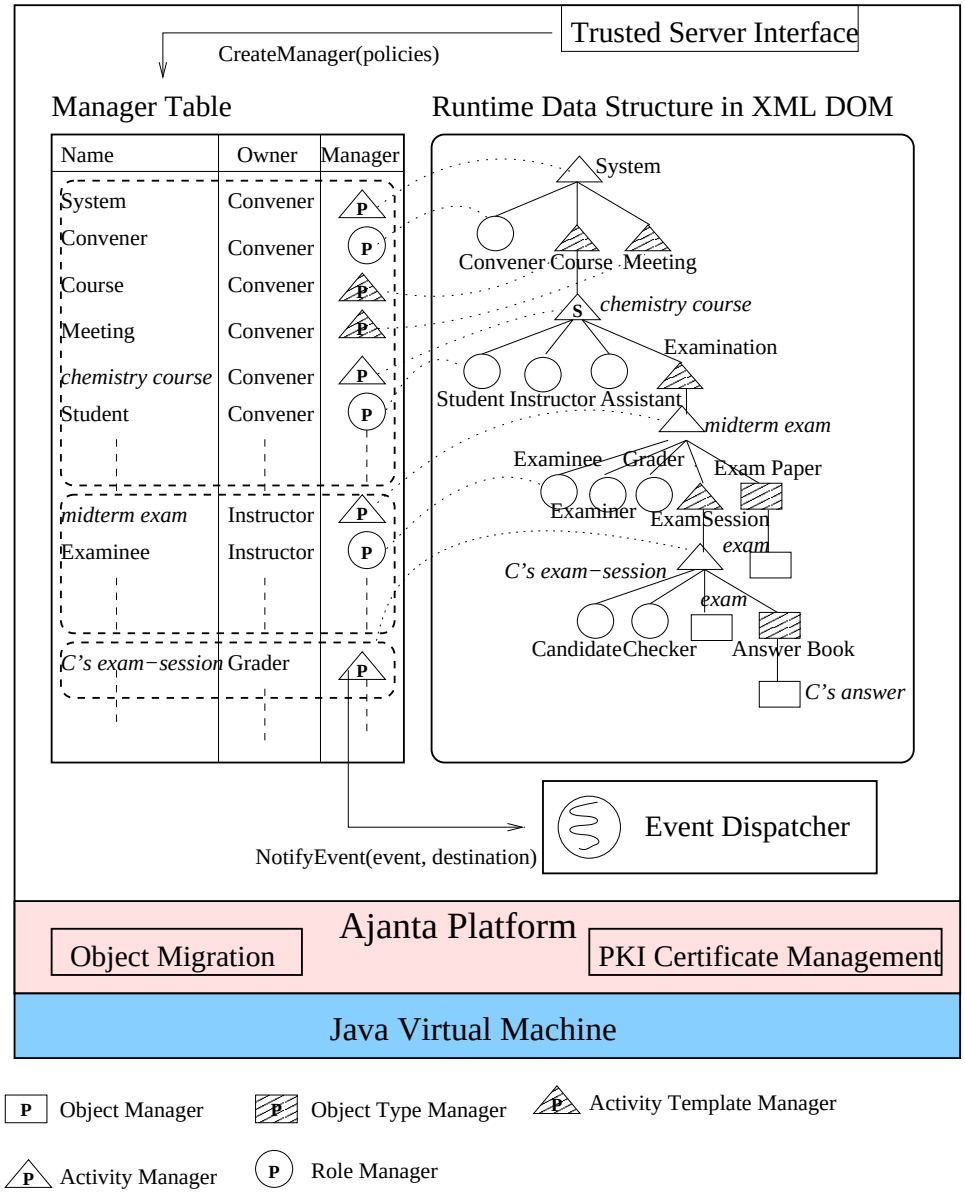

Figure 6.10. Trusted server components to manage protection domains

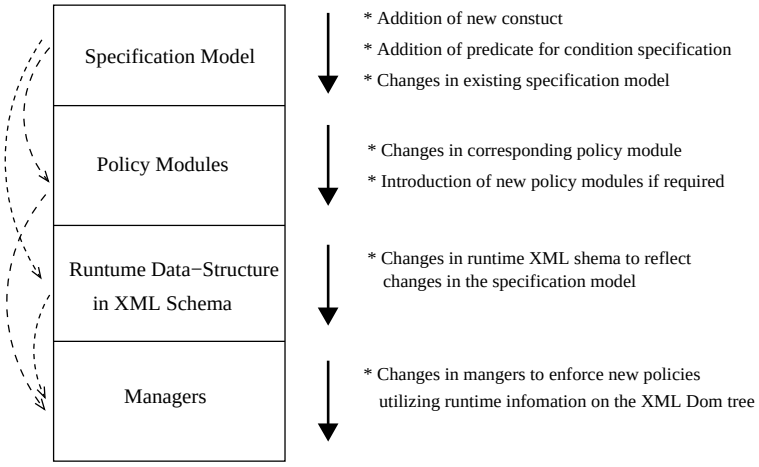

Figure 6.11. Steps in adapting changes in the collaboration specification model

created, corresponding DOM tree is expanded for the activity instance and references to entities in the DOM tree are passed to the newly created managers.

We explore the technique of maintaining the runtime data structure in XML for ease of adaptation of the middleware with the changes in the specification model. Maintaining runtime information of activity instances based on an XML schema enables automatic adaptation of the runtime data structure based on changes in the specification model that is also expressed in XML. A change in the specification model is incorporated by corresponding modification of the XML schema for the runtime data structure. The steps required to adapt to a change in the specification model are shown in Figure 6.11.

6.2.4. Interaction Models

As all tasks within our collaboration model are specified as role operations, a user may not need to know which objects are invoked as part of a role operation. To maintain the consistency of roles' state in the execution environment, we impose the restriction that all the interactions among users and objects must be initiated through

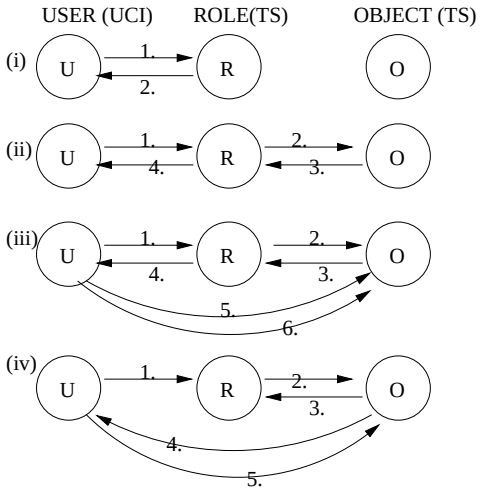

Figure 6.12. Interaction model

a role. A role operation can result in a method invocation, or initiation of a "user-object session" through which a user can have extended interactions with the object.

Figure 6.12 represents the interaction model in our collaboration system, where a user invokes a role operation (arrow 1), which may invoke an object (indicated by arrows 2, 3). Completion, failure, or initiation of a "user-object session" for this operation is notified to the user (arrow 4). If a "user-object session" is initiated, user can have multiple direct interactions with the object (arrow 5). [1]

6.3. Middleware Services

Following are the basic services provided by the middleware, which are required to securely manage the runtime environment.

[1]For simplicity of discussion related to the specification model and trust issues, we generalize that each operation only interacts with a single object. This generalization can be extended for multiple object interactions, as an adapter object can be constructed to handle issues related to multiple objects' method invocations.

6.3.1. Naming and Certificate Service

Ajanta's secure name service assigns a globally unique name (URN) to every entity within a collaboration system. *Identity Certificates* are issued by the name service, representing a mapping between an entity's name and its public key. Names in our URN based names are hierarchically structured. An instance of the *ExamSession* activity has the name:

"system.Examination.midterm.ExamSession.1".

Role managers issue *Membership Certificates* to all role members. All certificates are verified by the issuer.

6.3.2. Coordination Consistency Management

In our model, collaboration entities – activity, role, and object – may be managed at different nodes. In such situations, there is no single node that has a complete view of the system state. Our specification model, on the other hand, assumes a centralized view of the system state with serialized evaluation of operation preconditions[2]. In ensuring correctness in decentralized enforcement of coordination constraints, our goal is: *to allow only those sequence of operations that would be permitted by a centralized controller.*

Entities at different nodes may evaluate their operation preconditions concurrently. As some of these preconditions may be inter-dependent, inconsistencies may arise in precondition evaluation due to the following reasons:

- Events are communicated using either the *query (pull)* or *notification (push)* model. Due to delays in event notification, a role may have a stale view of the state of another entity on which its precondition depends.

- While a role's operation precondition is being evaluated, the state of other entities, on which it depends, may change.

[2]Evaluation of an operation's precondition and increment of its *start* event is atomic

Consider the following example, showing the specification of two role operations, only one of which may be performed.

```
Role r1: Operation op1
  Precondition #(op2.start) = 0 Action ....
Role r2: Operation op2
  Precondition #(op1.start) = 0 Action ....
```

With a centralized coordinator, the evaluation of the conditions is serialized ensuring that as soon as either of the operation is started, the precondition of the other operation becomes false. In case of decentralized management of roles, where *r1* and *r2* are managed on different nodes, delay in the notification of the *start* event may result in both roles evaluating their preconditions to true resulting in an inconsistent collaboration state. The problem is aggravated by the fact that preconditions can be complex with dependencies involving a large number of operations of other roles. Note that a role is always completely managed at one node. Therefore inter-dependencies among operations of the same role will not cause inconsistencies in precondition evaluation.

To enforce consistent precondition evaluation, we must ensure that when an operation's precondition evaluates to true, the states of other entities on which it depends, do not change concurrently in a way as to make it false. A simple solution to the consistency problem is to enforce mutual exclusion in precondition evaluation among all entities, allowing only a single entity to evaluate operation preconditions at a time. In our collaboration specification model, based in the scope rules discussed in Section 4.4.3, a precondition can only refer to operations that are in the same activity scope. That is, a precondition only depends on operations that are defined within the same activity instance. Based on this observation, a simple solution is implemented in our current prototype. In this implementation, the activity instance manager enforces mutual exclusion by serializing precondition evaluations.

However, this is restrictive as it limits concurrency in the system. Moreover, this solution requires an activity manager to enforce all operation event based condition

evaluations. These include evaluation of precondition for object access and intra-role coordination. Based on our trust model, the evaluation of precondition of object access is assigned to object managers. Similarly, enforcement of intra-role coordination within a role manager can only ensure secrecy of member identities. However, if the activity manager and the object or the role manager have the same owner, the trust of managing precondition evaluation can be delegated to the activity manager.

A less restrictive solution is that while a role is evaluating its precondition, we block only the evaluation of those operations' preconditions that may potentially affect its consistency. We refer to this set of operations as the *dependency set* of that operation. This set includes all those operations whose events appear in its precondition.

Our goal is to maximize concurrency in decentralized evaluation of coordination constraints while ensuring that the collaboration state is consistent. We therefore define the *minimal dependency set* for an operation as a subset of the *dependency set* containing only those operations that can change its precondition from true to false. An operation's minimal dependency set signifies the operations that must be blocked while its precondition is being evaluated.

To find this minimal dependency set for an operation, we identify those operations with respect to which the precondition is either *independently consistent* or *dependently consistent*, as described below:

Independently Consistent: The operation's precondition is said to be independently consistent with respect to an operation from its dependency set, if any events generated by that operation cannot change the precondition from true to false. Consider the following example of the bounded buffer problem with buffer size N:

```
Role producer:  Operation put
  Precondition #(put.start) - #(get.finish) < N
Role consumer:  Operation get
  Precondition #(put.finish) - #(get.start) > 0
```

The *put* operation's precondition cannot change from true to false as the count of

put.start cannot increase during its own precondition evaluation, and the *get.finish* count is monotonically increasing. In this case the *put* operation's precondition is *independently consistent* with respect to the *get* operation.

Dependently Consistent: If the precondition evaluates to true, and if it can be guaranteed that an operation on which it depends cannot concurrently change its state, we say the precondition is *dependently consistent* with respect to that operation. The following example specifies two operations that must be performed in strict alternation. Therefore, when the precondition of one operation is true, that of the other is false, ensuring that its state does not change while the first operation's precondition is being evaluated.

```
Role r1:  Operation op1
  Precondition #(op1.start) - #(op2.finish) = 0
Role r2:  Operation op2
  Precondition #(op1.finish) - #(op2.start) > 0
```

To arrive at the minimal dependency set of an operation, we remove from its dependency set those operations with respect to which its precondition is either independently or dependently consistent.

Similar to precondition evaluation, admission constraints may require blocking mechanisms when specifying such policies as "static separation of duties". Consider the following example where a user cannot join both the roles *r1* and *r2*.

```
Role r1:  Admission Constraints !member(thisUser, r2)
Role r2:  Admission Constraints !member(thisUser, r1)
```

Here, *r1*'s admission constraint depends on the user's membership in *r2*, and vice versa. To ensure that a user cannot join both the roles, when evaluating a role's admission constraints for a specific user, the evaluation of the other roles admission constraint for that specific user needs to be blocked.

To check the feasibilty of this approach, the theorem prover HOL (HOL, 2003) is utilized in conjunction with the high level specification expressed in XML to determine the minimal dependency sets for role operations. In addition to the preconditions, for each operations the following rule is added: `OpName.start <= OpName.finish`. Based on incremental deletion of precondition rules that are in dependency set, a minimal dependency set can be found.

The dependency set can be further pruned dynamically when a predicate in an operation precondition becomes *stable* at runtime. A predicate is stable when its value can not change any further. For example, once the predicate *#(op.start)<5* becomes false, it cannot become true as event counts are monotonically increasing.

6.3.3. Event Subscription and Notification Service

Correct enforcement of security policies depends on the integrity of the events exchanged among the distributed entities. The middleware derives two event related policies from the collaboration specification for all collaboration entities. *Event subscription policy* specifies the valid subscribers of an event generated by an entity. *Event notification policy* specifies the valid notifier for each subscribed event. Authenticating the notifier cannot guarantee that the notifier is not *falsifying* or *omitting* events. That is, the notifier may not be trusted for a received event or may not send an event when it may be to its advantage.

In Chapter 5.3, as part of the verification mode with untrusted owner, the *falsifying* or *omitting* events are modeled. In this section, we investigate the security mechanisms in the event service that prohibits untrusted owners to *falsify* or *omit* events. In this section, we present a solution, if implemented, does not require trust on the owner for not manipulating event causality. In our middleware, event communication is based on Java RMI, which ensures synchronous delivery of events.

Event Generation: The issue related to event generation is: which entity can

be trusted to generate an event. When a role operation does not invoke an object method, only the role can generate the *start* and *finish* event for the operation. When the operation accesses an object, either the role can generate the *start* event after confirmation from the object or the object can generate it. The *finish* of the operation is generated by the object when a "user-object session" has been initiated. Following scenarios show when events received from an entity cannot be trusted without corroborating events.

Example 1 A *start* or *finish* event generated by a role operation, which invokes an object method, can be *falsified*. In the following example, operation *op1* invokes an method on the object *obj1*. Participants of role *r2* can perform operation *op2*, only if *op1* has been started.

```
Role r1:  Operation op1 Action obj1.method()
Role r2:  Operation op2 Precondition #(op1.start) > 0
```

Consider roles *r1*, *r2*, and object *obj1* are maintained by distinct owners. Role *r1* can generate event *#(op1.start)=1*, even though *obj1* has declined to perform the action associated with *op1*. If the responsibility of generating *start* event is assigned to the object, the issue still remains the same – *obj1* can generate the event without *r1* performing *op1*.

Example 2 In our model, role operation related events can be causally dependent. An untrusted role may falsely generate an event thereby violating such causality. In the following example, role *r2's* operation *op2* depends on *r1's* role operation *op1*, and the operation *op3* can be performed only if *op2* has been performed once.

```
Role r1:  Operation op1 Precondition ....
Role r2:  Operation op2 Precondition #(op1.start) = 1
Role r3:  Operation op3 Precondition #(op2.start) = 1
```

Here, *r2* may send *r3* the event *#(r2.op2.start)=1* even though *op2's* precondition has not been satisfied.

Example 3 When there are more than one subscribers for an event, the notifier may strategically omit sending the event to some targeted subscribers for its own advantage.

Ensuring Trust in Event Communication: Here, we present solutions for the above three cases [3]: The security issue in Example 1 is introduced because the subscriber trusts a single entity for the *start* event, when the event actually signifies two facts: the role has invoked the operation and the object has started it. In our middleware, for a role operation with an associated object action, subscribers receive a *request* event from the role and a *start* event from the object. All operations requested may not be started resulting in more *request* events than *start* events. The general problem for the subscriber is to find that there is a *request* event corresponding to the operation's *start* event. This correspondence is facilitated by the middleware, which supports signed event notification. In the case of *finish* events, a similar problem arises, which is solved by subscribing corroborating *start* events.

The problem in Example 2 occurs as role *r3* wrongfully trusts role *r2* for enforcing its operation *op2's* precondition. In our middleware, if such trust cannot be conferred, a role enforces all other role operations' preconditions on which its precondition depends. In Example 2, consider that *r1*, *r2*, and *r3* are in different trust domains. Then *r3* has to subscribe: events on which *op2's* precondition depends, i.e., events from *op1*; any events on which *op1's* precondition depends; and so on. However, if *r1* and *r2* are in the same trust domain, *r3* does not need to subscribe events related to *op2's* precondition.

The third problem of tactical event omission, as discussed in Example 3, is comparatively harder to solve in an efficient manner, however, it can be solved by any distributed agreement protocols. If there are more than one subscribing domains for an event, they need to agree on a common trusted notifier. Our owner assignment rules create a trust hierarchy among the entities in the collaboration. In the hierarchy, as an entity trusts its immediate predecessor, for a given set of entities, there is

[3]Current prototype implementation has not adopted these solutions

always an entity, which can be entrusted with this specific responsibility.

6.3.4. Protocols for Secure Interactions

The protocols for secure interactions among the various entities in the collaboration is developed based on the basic challenge response protocol (Needham and Schroeder, 1978). Ajanta ticketing service (Karnik and Tripathi, 2001) is extended with a cascaded tickets model for multi-party - user, role, and object managers – authentication. Detail design of the implemented protocols can be found in M.S. Plan B report of Richa Kumar (Kumar, 2002).

6.4. Functional Design

In this section, the design of the main components, i.e., managers of the middleware is discussed. The functional aspects of different managers as well as their interactions are presented.

6.4.1. Generic Managers

The classes representing the managers in the middleware are extended from a generic manager class. The components for the generic manager is shown in Figure 6.13. The manager class implements four remote interfaces: `Manager`, `Subscriber`, `Notifier`, and `Authenticator`.

- `Manager` interface is for querying meta-information of a manager object – such as the name, owner, and creator. It also supports a meta-method for instantiating a manager object required for two-phase protocol of instantiating a manager.

- `Subscriber` interface is for authenticated notification of events.

- `Notifier` interface is for authenticated subscription of events.

- `Authenticator` interface is for initiating a challenge in challenge-response protocol for authenticated communication with a manager.

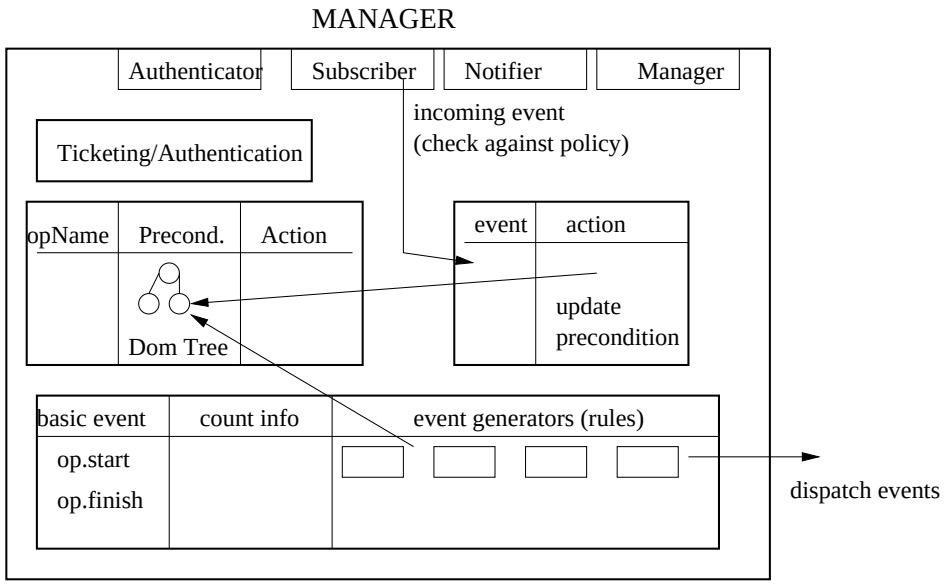

Figure 6.13. Generic manager design

The primary functionality supported by a generic manager is to enforce *Event Subscription/Notification Policy* and `Ticketing` for authenticated communications. All the manager classes are extended from a generic manager as shown in Figure 6.14.

Role Manager

Each role manager implements a remote `RoleManager` interface. This interface supports two distinct types of functionalities. First functionality related to the role membership, operations, and trusted server selection. These are:

- Interface for generic role membership operation, such as users' join, leave, admit, and remove.
- Interface to manage membership related to role reflection.
- Interface for querying role membership information, such as listing members of the role, checking if a user is a member of the role, and counting members of the role.

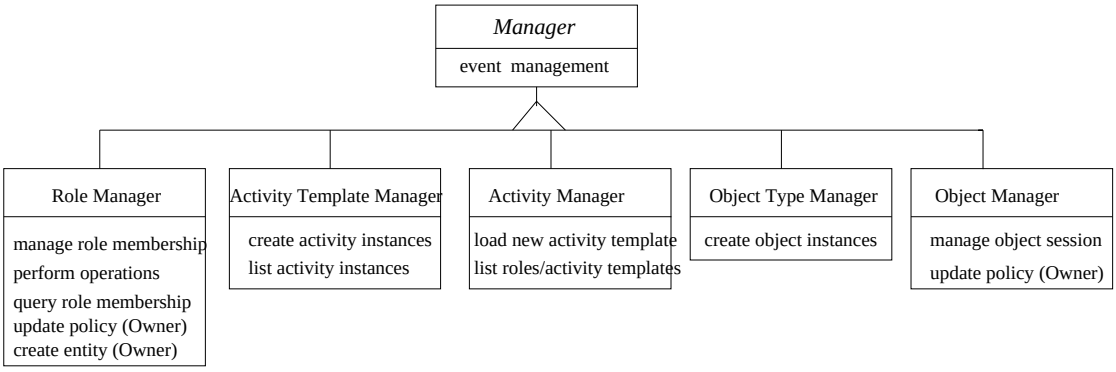

Figure 6.14. Managers hierarchy

- Interface for querying specified role operations and invocation of such an operation.

- Interface for setting a trusted server for the role.

The second set of interface is required when this role is an owner and is obliged to create other entities and update policies for entities it manages. In our specification model, owner roles are selected from the same or outer activity scope. When the outer activity in created the owner role is provided with policies that specifies the the entities it will manage in the inner activity scope. When the entities in the inner activity scope is created this owner role is contacted through these interfaces for either to create the entities or to update the policies.

Activity Template and Activity Manager

Primary functions of an activity template is to create activity instances. `ActivityTemplateManager` interface can be queried for listing of its instances. The `ActivityManager` Interface can be queried for its roles and nested activity templates. The UCI utilizes these interfaces to traverse the activity instances given a trusted server.

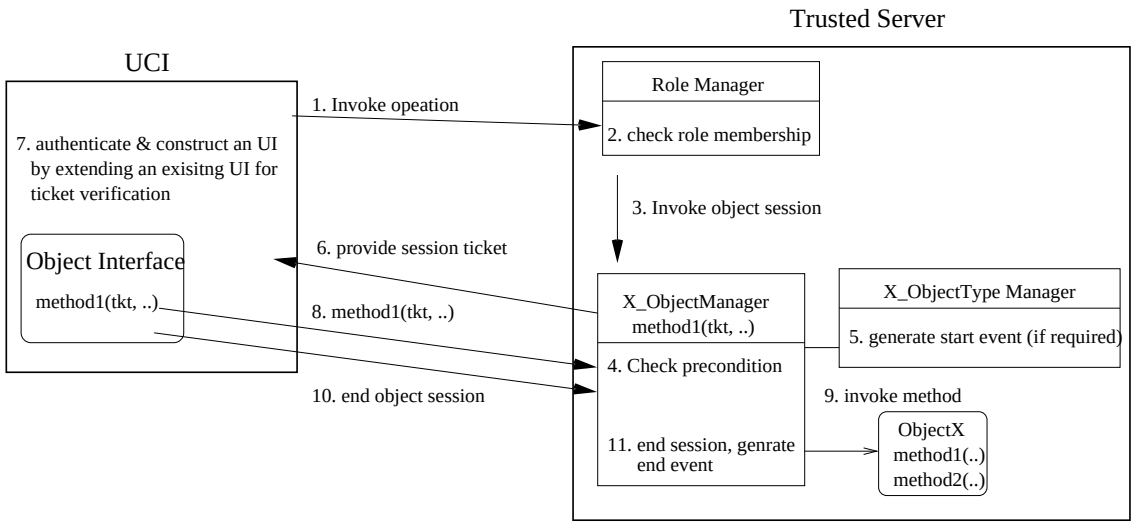

Figure 6.15. User interaction with an object through managers

Object Type and Object Manager

The policy update is required as in multiple scopes of nested activities, the number of activity scope between the owner role and an entity it owns can be more than one. As an intermediate activity instance is created, the policy in the owner role is updated for proper binding.

For interaction with an object initiated through an operation, a *UserInterface* class can be assigned. However, users are not restricted to a specific *UserInterface* class and these *UserInterface* classes are not part of the specification model.

6.4.2. Activity Management

In our model activities are created either by installing a new *activity template* and instantiating it or by instantiating a nested activity through a role operation. The installation of an *activity template* and its subsequent instantiation is usually performed by the *Convener* on its trusted server. From the security point of view, this activity creation protocol is simple as all the steps in the protocol are performed on the *Convener's* trusted server. However, when a nested activity is instantiated,

the activity creation and distribution protocol has to manage several security aspects:

- The role manager, which initiated the nested activity creation, contacts the owner of the nested activity to create the new activity instance's manager.
- The initiating role manager proves to the role manager of the new activity's owner that it is authorized to create the requested activity.
- Similarly, if the activity has encapsulated roles, the role managers of the owners of these roles contact and provide with the certificates that the initiating role manager has the rights to create the corresponding role managers.
- The new activity and its nested roles are created on their owners' trusted servers.
- If users are assigned to roles as part of activity instantiation, the new role managers update the users' UCIs with the user specific views of the activity.

In our model, a predefined order of object accesses occur during generation of application defined events.

6.5. Related Work: Policy Driven Construction

Many other systems (D. Wodtke and Kotz-Dittrich, 1996; Dewan and Shen, 1998a) have introduced a generic middleware for implementing workflow and specific CSCW applications. However, only few of the these systems support policy-driven architecture. In workflow specification initiative, Wfmc (Workflow Management Coalition, 1999) is concerned with interoperability of workflow systems and mainly focuses on data exchange related issues.

A wide range of systems address runtime selection of concurrency or coordination policies in synchronous shared-space based CSCW systems. Share (Greenberg, 1991), a shared screen system, is a personalizable groupware, which offers users several choices of floor control models. It provides primitives to program new floor control policies. Groupkit (Roseman and Greenberg, 1996; Roseman and Greenber, 1993) is a groupware toolkit for building synchronous group conferencing. It supports open

protocol to implement dynamic floor control policies. On the other hand, GROVE (GRouping Outline Viewing Editor) (Ellis and Gibbs, 1989; Ellis et al., 1991), a realtime test editor for a group of people, which allows concurrent update of texts by users implemented a concurrency control mechanism, termed *Operational Transformation*, which is a dependency-detection solution with automatic conflict resolution. Rendezvous (Hill et al., 1994) is a architecture and a programming language to build multiuser applications for realtime collaboration written in Lisp and CLOS, and uses event-based scheduling to process user input. It introduces an Abstraction-Link-View (ALV) structure compared to traditional Model-View-Controller(MVC) for user interface management. In ALV, links are constraints for value consistency of abstraction and views, and the constraints are declarative specification.

In (Guruduth Banavar and Mukherjee, 98), a component-based Java Beans supported suite is introduced for rapidly building synchronous collaborative applications. This requires a middleware to be implemented using the component suite, which is limited to current Java Beans tools (Guruduth Banavar and Mukherjee, 98). However, this approach does not support any specification facility for runtime construction of an application. COLA (Jonathan Trevor, 1993; Trevor et al., 1994) presents techniques of realizing shared object service by augmenting existing objects through object adaptors. For dynamic access control policy enforcement, each client context including a name, a role, and an activity condition are passed to object adapters. This is checked against a ladder and rule set to validate the requested operation. The activity presented in this paper is termed as lightweight activity, which is used to capture the stages the activity moves through to specify dynamic access control. The lightweight activity is contrasted with traditional activity model which tries to capture the dependencies between sub-tasks and the behavior exhibited by different roles.

In COCA(Li and Muntz, 1998; Du Li and Muntz, 1999; Li and Muntz, 1999), a role is a set of rule-based actions executed by a participant. There is no notion of policies for access control and role admission. COCA supports dynamic role definition change

through adding new rules. But it is not clear how one can easily specify security policies, such as who can remove an object from collaborative workspace after the object has gone through changes, or how one can specify separation of duty constraints for a role for COCA's strong tie to IP multicast models. COCA specifies policy for only interactive applications. There is no security on event propagation to specify which user can see what events. COCA (Li and Muntz, 1998) provides a logic-based specification language for coordination policies separating specification of coordination from computation. However, COCA does not address security requirements either in specification or in system implementation, and it is not clear how security policies can be integrated with its coordination policies as in the implementation level it is strongly tied to IP multicast models and Prolog.

DCWPL (Corts and Mishra, 1996) is a coordination language for CSCW applications, which addresses user level mechanism to deal with group interaction issues. A DCWPL program holds control information for artifacts, e.g. text and functions. However, no support is provided to check or prevent the program to be in an inconsistent state. Here, roles are viewed as data-types to characterize a set of artifacts and a set of functions. However, the example specification uses predefined roles to create new roles, and it is not clear how new predefined roles can be added. If a policy or function is not available in declarative DCWPL, the user still needs to program it in a procedural language (Li and Muntz, 1998).

6.6. Summary

In this chapter, we discussed the design of a policy driven middleware that supports construction of secure distributed collaboration environments from high level specification of coordination and security policies. The middleware is developed as a proof of concept implementation.

Chapter 7

Experimentation and Evaluation

A motivation of our research is to realize collaboration environments based on different sets of coordination and security polices using the same set of shared resources. In this chapter, using example collaboration environments, we evaluate two aspects of our approach. First, we show that our specification model can specify and the middleware can construct different collaboration environments using the same shared resources. Second, we discuss the expressiveness of the specification model for specifying different coordination and security requirements.

In this chapter, four different collaboration environments are presented. These are *Whiteboard Sharing*, *Audio Sharing*, *Collaborative Notes Taking*, and *Collaborative Document Authoring*. More than one set of requirements are used for *WhiteBoard* and *Audio* sharing to show how different sets of coordination and security policies can be expressed on the same object. The *Collaborative Notes Taking* and and *Collaborative Document Authoring* examples are utilized to show the expressiveness of the specification model.

The *WhiteBoardSharing* examples show how a collaboration environment can be extended to include new sets of requirements. These examples also show that the shared object may also need to be extended to support the interfaces based on the collaboration designer's requirements. The *Audio Sharing* example shows how secure

enclaves can be constructed based on the administrative model of the specification. The *Collaborative Notes Taking* example is present task-flow, role-constraints, information flow and access leakage related requirements in a collaboration environments. Lastly, the *Collaborative Document Authoring* example presents a collaboration with multiple nested activity templates with concurrent activity instances.

7.1. Whiteboard Sharing

In this section. we show that a *WhiteBoard* can be shared based on different coordination requirements. Here, we discuss the following four types of policies. These policies are selected to present the expressiveness and limitations of the specification model.

1. Un-restricted sharing of a *WhiteBoard* where any participant can draw on the board.

2. Moderator controlled white board sharing, where a moderator grant the participant who has least recently requested to draw on a *WhiteBoard*.

3. Moderator control the board sharing and can grant the board to any requesting attendee. Moreover the moderator can revoke the board from an attendee.

4. Attendees vote to select the next participant who can draw on a board. Various forms of voting policies can be selected. In this example, all the attendees must vote in each round before a candidate can be selected.

7.1.1. Un-Restricted Sharing

Following are the coordination and security requirements for uncoordinated use of a *WhiteBoard* application.

- *WhiteBoard* can be shared by all the users within a *WhiteBoardSharing* activity.

- Any user can join a *WhiteBoardSharing* activity.

The method signatures for a *WhiteBoard* object in presented below. The *draw* method enables drawing on the board. The *addViewer* method registers a *ViewListener* object with the *WhiteBoard* so that any update on the board can be propagated to the corresponding graphical interface.

```
public class WhiteBoard{
  public void draw (DrawInstruction di);
  public void addViewer (ViewListener vl);
  public void removeViewer (ViewListener vl);
}
```

The specification for the *WhiteBoardSharing* activity based on the requirements of uncoordinated sharing is presented in Example 1. In the example, the *WhiteBoardSharing* activity template has only one role *Attendee* (line 02). Any user can join this role. After a member of the *Attendee* role creates the *WhiteBoard* object (line 03), all the member of the *Attendee* role can *draw* on the board (line 08). The *OpenUI* operation grants an attendee view permission for the board (Line 13).

Example 1 WhiteBoard sharing: uncoordinated sharing

```
01 ActivityTemplate WhiteBoardSharing () {
02   Role Attendee {
03     Operation InitiateWhiteBoard
04       Precondition
05         #(InitiateWhiteBoard.start) = 0
06       Action
07           whiteboard = new Object(WhiteBoard)
08     Operation Draw
09       Precondition
10         #(InitiateWhiteBoard.finish) > 0
11       Action
12         Grant whiteboard {void draw(DrawInstruction di)}
13     Operation OpenUI
14       Precondition
10         #(InitiateWhiteBoard.finish) > 0
15         ∧ #(OpenUI.start(invoker=thisUser)) = 0
16       Action
17         Grant whiteboard {void addViewer (ViewListner vl)}
18   }
19 }
```

7.1.2. Moderator Control: First-Requested First-Granted

Following are the coordination and security requirements for coordinated use of a *WhiteBoard* object.

- *Moderator* controls the sharing of the *WhiteBoard* object.
- Participants of the *Attendee* role requests to access the board.
- *Moderator* assigns control to a requesting attendee in a first-requested first-granted manner.
- Once granted an attendee can access the board until he/she finishes his/her task of drawing on the board.
- An attendee can only request, if he/she does not have any outstanding request.
- The moderator can only grant if (1) there is an outstanding requests, and (2) the board is not currently under use.
- The *Draw* operation can only be invoked when (1) the same invoker is not currently drawing on the board, and (2) there is an outstanding *grant* for the invoker.

In this example, to keep track of the attendees who have requested the board a *Queue* object is used. The method signature of this object is shown below. The *get* method returns a passed object in a first-in first-out manner. The *setFirst* method sets the passed object in front of the queue.

```
public class Queue {
  public String get( );
  public void put (String s);
  public Enumeration elements ();
  public void remove (String s);
```

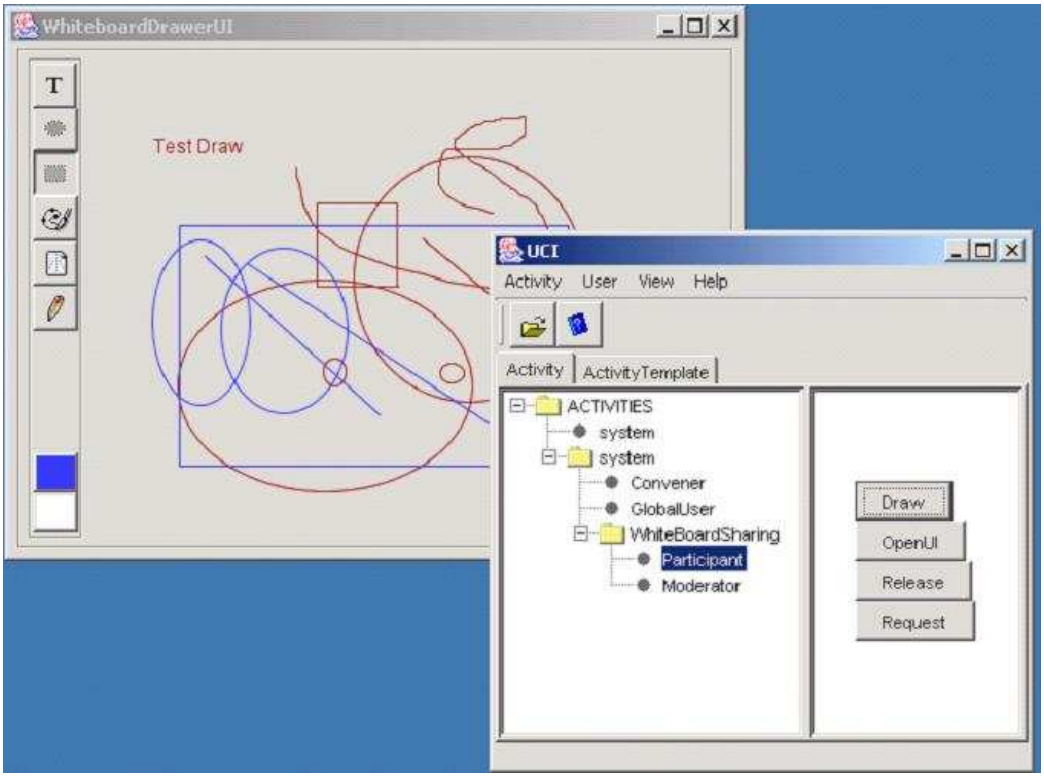

Figure 7.1. Operations in UCI for whiteboard sharing

```
    public void setFirst (String s);
}
```

The specification for *WhiteBoardSharing* activity based on the above requirements is presented in Example 2. In this *WhiteBoardSharing* activity template, there are two roles: *Moderator* and *Attendee* (line 02 and 17). A member of the *Moderator* role has to be assigned when an *WhiteBoardSharing* activity is created (line 01). Any user including the moderator can join the *Attendee* role.

Example 2 WhiteBoard Sharing: first-requested first-granted

```
01 ActivityTemplate WhiteBoardSharing (AssignedRoles Moderator) {
```

```
02  Role Moderator {
03    AdmissionConstraints #members(thisRole) < 1
04    Operation InitiateWhiteBoard
05      Precondition
06        #(InitiateWhiteBoard.start) = 0
07      Action
08          queue = new Object(Queue)
09          whiteboard = new Object(WhiteBoard)
10    Operation Grant
11      Precondition
12        #(Attendee.Request.start) - #(Grant.start) > 0
13        ∧ #(Grant.start) - #(Attendee.Draw.finish) = 0
14    Action
15      NotifyEvent AssignControl( grantee = queue.get());
16  }
17  Role Attendee {
18    Operation Request
19      Precondition
20        #(InitiateWhiteBoard.finish) > 0
21        ∧ #(Request(invoker=thisUser).start) −
22          #(Moderator.AssignControl(grantee=thisUser)) = 0
23        ∧ #(Moderator.AssignControl (grantee=thisUser)) −
24          #(Draw(invoker=thisUser).finish) = 0
25      Action
26        InvokeMethod queue.put(thisUser)
27    Operation Draw
28      Precondition
29        #(InitiateWhiteBoard.finish) > 0
30        ∧ #(Draw(invoker=thisUser).start) −
31          #(Draw(invoker=thisUser).finish) = 0
32        ∧ #(Moderator.AssignControl(grantee=thisUser)) −
33          #(Draw(invoker=thisUser).start) > 0
34      Action
35        Grant whiteboard {void draw(DrawInstruction di)}
36    Operation OpenUI
37      Precondition
38        #(OpenUI.start(invoker=thisUser)) = 0
39      Action
40        Grant whiteboard {void addViewer (ViewListner vl)}
41  }
42 }
```

In the figure, a member of the *Attendee* role can invoke the *Request* operation (line 18), if he/she does not have an outstanding request, i.e., the number of *Request* and the number of *AssignControl* for the invoker is equal (line 21-22) . Moreover, an

attendee can only *Request*, if he/she is not currently drawing on the board, i.e., the number of *AssignControl* and the number of finished *Draw* operation for the invoker is equal (line 23-24). As part of requesting to access the board, the attendee puts his/her identity in the *Queue* object (line 26).

The moderator can only *Grant* (line 10) when there is an outstanding request, i.e., the number of the *Request* operation is larger than the number of the *Grant* operation (line 12). Moreover, the *Grant* operation can only be performed when the board is not accessed by an attendee, i.e., the number of *Draw* operation that has been finished is equal to the number of the *Grant* operation (line 13). The action of the *Grant* operation generates an *AssignControl* event with the attribute *grantee* with the name of the attendee who has least recently requested the board (line 15). This event enables other operations to coordinate on the value of the assigned grantee.

The *Draw* operation (line 27) can only be invoked when the invoker is not currently drawing on the board, i.e., the number of the start and the number of the finish of the *Draw* operation is equal (line 30 -31). Also an attendee can only *draw*, when there is an outstanding *Grant* for the invoker, i.e., the number of *AssignControl* for this attendee is larger than the number of *Draw* operation invoked by this attendee (line 32-33).

In this example, the *Moderator* role creates a *WhiteBoard* and a *Queue* object (line 08-09). These objects cannot be accessed until the operation *InitiateWhiteBoard* to create these objects is finished (line 20 and 29).

7.1.3. Moderator Control: Moderator Select and Revoke

The previous example of *WhiteBoradSharing* is extended with the following requirements in this Section.

- The moderator can select any attendee among the requesting users as grantee to draw on the board.
- The moderator can revoke, from an attendee, the access-right of drawing.

In our current middleware, the *ObjectManager* provides an interface *endOperationSession* for termination of an object-session initiated through an operation. The invoker of an operation is provided with a *ticket* that authorizes the operation invoker to terminate the object-session (Kumar, 2002). Any other user cannot terminate an object-session. For termination of an object-session by other role, object level methods are implemented [1]. For the requirement that the *Moderator* can revoke the *draw* permission from an attendee, we have extended the *WhiteBoard* object with two additional methods: *revokeDraw* and *enableDraw*. These methods are shown in the following method signatures of the *WhiteBoard* object.

```
public class WhiteBoard{
  public void draw (DrawInstruction di);
  public void addViewer (ViewListner vl);
  public void removeViewer (ViewListner vl);
  public void revokeDraw ();
  public void enableDraw ();
}
```

The extended specification for *WhiteBoardSharing* activity based on the new requirements is presented in Example 3. In this example two additional operations are added: *Revoke* operation for the *Moderator* role (line 17) and *Release* operation for the *Attendee* role (line 35). These two operations invoke the *revokeDraw* method on the board to stop drawing (line 21and 41). The drawing in enabled with the *Draw* operation (line 51).

In this example, a cycle of *Request*, *Grant*, and *Release*/*Revoke* operations can be performed. An attendee can *Request* only if he/she does not have an outstanding grant. That is the number of *AssignControl* for the invoker is equal to the addition of the number of the *Release* and *RevokeControl* for the same user (line 30-32). Similarly. the moderator can *Grant* only if the control from the board is either revoked

[1] This can be avoided with specification and middleware level support

or released. That is the total number of *Revoke* and *Release* is equal to current *Grant* (line 13). In the action of the *Grant* operation, the moderator can manipulate the *Queue* object to select any participant from all the requesting attendees (line 15).

Example 3: WhiteBoard sharing: moderator select and revoke.

```
01 ActivityTemplate WhiteBoardSharing (AssignedRoles Moderator) {
02   Role Moderator {
03     AdmissionConstraints #members(thisRole) < 1
04     Operation InitiateWhiteBoard
05       Precondition
06         #(InitiateWhiteBoard.start) = 0
07       Action
08           queue = new Object(Queue)
09           whiteboard = new Object(WhiteBoard)
10     Operation Grant
11       Precondition
12         #(Attendee.Request.start) - #(Grant.start) > 0
13         ∧ #(Grant.start) - #(Attendee.Release.start) - #(Revoke.start) = 0
14     Action
15       Grant queue { get(); elements(); remove(); setFirst() }
16       NotifyEvent AssignControl( grantee = queue.get());
17     Operation Revoke
18       Precondition
19         #(Grant.start) - #(Attendee.Release.start) - #(Revoke.start) > 0
20     Action
21       InvokeMethod whiteboard.revokeDraw()
22       NotifyEvent RevokeControl( revokee = queue.get());
23   }
24   Role Attendee {
25     Operation Request
26       Precondition
27         #(InitiateWhiteBoard.finish) > 0
28         ∧ #(Request(invoker=thisUser).start) −
29           #(Moderator.AssignControl(grantee=thisUser)) = 0
30         ∧ #(Moderator.  AssignControl (grantee=thisUser).start) −
31           #(Release(invoker=thisUser).start) −
32           #(Moderator.RevokeControl(revokee=thisUser)) = 0
33       Action
34         InvokeMethod queue.put(thisUser)
35     Operation Release
36       Precondition
37         #(Moderator.  AssignControl (grantee=thisUser)) -
38         #(Release(invoker=thisUser).start) -
39         #(Moderator.RevokeControl(revokee=thisUser)) > 0
```

```
40      Action
41        InvokeMethod whiteboard.revokeDraw()
42    Operation Draw
43      Precondition
44        #(InitiateWhiteBoard.finish) > 0
45        ∧ #(Draw(invoker=thisUser).start) −
46          #(Draw(invoker=thisUser).finish) = 0
47        ∧ #(Moderator.AssignControl(grantee=thisUser)) −
48          #(Release(invoker=thisUser).start) −
49          #(Moderator.RevokeControl(revokee=thisUser)) > 0
50      Action
51        InvokeMethod whiteboard.enableDraw()
52        Grant whiteboard {void draw(DrawInstruction di)}
53    Operation OpenUI
54      Precondition
55        #(OpenUI.start(invoker=thisUser)) = 0
56      Action
57        Grant whiteboard {void addViewer (ViewListner vl)}
58  }
59 }
```

7.1.4. Voting

In this section, the Example 2 of *WhiteBoradSharing* is extended with the following requirements.

- The attendees vote to select the next candidate who can draw.
- The moderator assigns control to the selected candidate only after all the attendees have voted.

In this example, instead of a *Queue* object, a *Vote* object is used as shown below. In the *Vote* object, *put* adds a user as a candidate to be selected by voting; *elements* returns all such candidates; *vote* assigns a vote for a candidate; and *result* returns the candidate with the largest votes. *result* always returns a single candidate and clears all the votes.

```
public class Vote {
  public void put (String s);
```

```
    public Enumeration elements ();
    public String result ();
    public vote (String voter, String candidate);
  }
```

Compared to Example 2, in Example 4, a *Vote* operation is introduced for the *Attendee* role (line 36). An attendee can only vote if (1) there is an outstanding request (line 38), (2) no outstanding grant (line 39), and (3) all the votes by this invoker have already been used in earlier voting rounds (line 40). Similarly, the moderator can only *Grant* when all the attendees have voted (line 13). That is there is an extra round of votes available.

Example 4: WhiteBoard sharing: voting to select the next participant to draw.

```
01 ActivityTemplate WhiteBoardSharing (AssignedRoles Moderator) {
02   Role Moderator {
03     AdmissionConstraints #members(thisRole) < 1
04     Operation InitiateWhiteBoard
05       Precondition
06         #(InitiateWhiteBoard.start) = 0
07       Action
08           vote = new Object(Vote)
09           whiteboard = new Object(WhiteBoard)
10     Operation Grant
11       Precondition
12         #(Attendee.Request.start) - #(Grant.start) > 0
13         ∧ #(Grant.start) - #(Attendee.Draw.finish) = 0
13         ∧ #(Vote.finish) - (#(Grant.start + 1)*#member(Attendee)) = 0
14     Action
15       NotifyEvent AssignControl( grantee = vote.result());
16   }
17   Role Attendee {
18     Operation Request
19       Precondition
20         #(InitiateWhiteBoard.finish) > 0
21         ∧ #(Request(invoker=thisUser).start) −
22           #(Moderator.AssignControl(grantee=thisUser)) = 0
23         ∧ #(Moderator.AssignControl (grantee=thisUser)) −
24           #(Draw(invoker=thisUser).finish) = 0
```

```
25        Action
26          InvokeMethod vote.put(thisUser)
27      Operation Draw
28        Precondition
29          #(InitiateWhiteBoard.finish) > 0
30          ∧ #(Draw(invoker=thisUser).start) −
31            #(Draw(invoker=thisUser).finish) = 0
32          ∧ #(Moderator.AssignControl(grantee=thisUser)) −
33            #(Draw(invoker=thisUser).start) > 0
34        Action
35          Grant whiteboard {void draw(DrawInstruction di)}
36      Operation Vote
37        Precondition
38          #(Attendee.Request.start) - #(Grant.start) > 0
39          ∧ #(Grant.start) - #(Attendee.Release.start) = 0
40          ∧ #(Vote(invoker=thisUser).start) - #(Grant.start) =0
41        Action
42          Grant vote { elements(); vote(String, string) )}
43      Operation OpenUI
44        Precondition
45          #(OpenUI.start(invoker=thisUser)) = 0
46        Action
47          Grant whiteboard {void addViewer (ViewListner vl)}
48    }
49 }
```

7.2. Audio Sharing

In this section, we present collaboration environments that share media resources. For sharing of audio streams, we use the same set of requirements presented in Section 7.1 for *WhiteBoard* sharing.

The method signatures for a *StreamCoordinator* object in presented below. A *StreamCoordinator* object receives audio stream from the registered senders and forward these streams to registered receivers. The *registerToSend* and *registerToReceive* methods enable senders and receivers to register with a *StreamCoordinator* object.

```
public class StreamCoordinator{
  public void registerToSend ( Object obj);
  public void registerToReceive (Object obj );
```

}

In our experimentations, we have developed collaboration environments for audio sharing based on the following requirements:

1. Uncoordinated floor control where all the participants can speak and listen.
2. Moderator based floor control where moderator decide on the next speaker based on different policies, such as assigning the floor to the least recently requesting user.
3. Voting based floor control where attendees vote to select the next speaker.
4. Secure audio sharing where an attendee can initiate a secure channel with other users of his/her choice.

In the above listing, the first three requirements and their corresponding specification are almost same as the example specifications of *WhiteBoard* sharing presented in the previous section. As the only difference between these two sets of specifications is the object type, we do not present them in this thesis. The last requirement, on the list, of secure *AudioSharing* in presented below in Example 5. In this example, *AudioSharing* activity template has the role *Attendee* (line 02) and a nested activity template *SecureChannel* (line 17). In an *AudioSharing* activity, an attendee can create a *StreamCoordinator* object (line 03-07). Once created, all the attendees can register to this object, to speak and to listen, by invoking the *OpenUI* operation (line 08-12). The nested *SecureChannel* activity template has only two roles: *Moderator* and *Speaker*. An attendee who creates an instance of the nested *SecureChannel* activity becomes the owner of the activity (line 18). He/she can select members from the *Attendee* role to assign to the *Speaker* role for the activity he/she owns (line 16). Being the owner of an *SecureChannel* activity, the attendee is also the moderator and creates an *StreamCoordinator* object within his/her protection domain. In the *SecureChannel* activity of an attendee, the attendee can add and remove other attendees to the *Speaker* role and has exclusive administrative privileges on the *StreamCoordinator* object.

Example 5 Secure private voice channels

```
01 ActivityTemplate AudioSharing ( AssignedRoles Attendee ) {
02   Role Attendee {
03     Operation createMediaStreamCoordinator
04       Precondition
05         #createMediaStreamCoordinator = 0
06       Action
07         stream = new Object(StreamCoordinator)
08     Operation OpenUI
09       Precondition
10         #createMediaStreamCoordinator > 0
11         ∧ #(OpenUI(invoker=thisUser).start) = 0
12       Action
13         Grant stream { registerToSend registerToReceive }
14     Operation CreateSecureChannel
15       Action
16         act = new Activity SecureChannel ( Moderator = thisUser, Speaker={ } )
17   }
18   ActivityTemplate SecureChannel (AssignedRoles Moderator, Speaker Owner Creator) {
19     Role Moderator {
20       AdmissionConstraints member(thisActivity.Creator)
21       Operation initiateStreamCoordinator
22         Precondition
23           #initiateStreamCoordinator = 0
24         Action
25           stream = new Object(StreamCoordinator)
26     Role Speaker {
27       AdmissionConstraints member(parentActivity.Attendee)
28       Operation OpenUI
29         Precondition
30           #initiateStreamCoordinator > 0
31           ∧ #(OpenUI(invoker=thisUser).start) = 0
32         Action
33           Grant stream { registerToSend registerToReceive }
34     }
35   }}
```

7.3. Collaborative Notes Taking

In this section, we present an online meeting collaboration. This example is adopted from Landay and Davis (1999). An online meeting activity has three roles: *Coordinator*, *Secretary*, and *Attendee*. A member of the *Coordinator* role is also an attendee. However, none of the members of the *Attendee* role can be a member of the *Secretary* role. After *Coordinator* initializes a *BulletinBoard*, members of *Attendee* can access,

i.e., read/write on the board. *Secretary* role can have only one member, who can only view the activities on the BulletinBoard and is responsible to scribe. Any attendee can initiate a private session of notes sharing inviting or allowing other selected attendees to view his/her notes. They may also submit his/her notes to the secretary. The notes-sharing activity is defined as a nested activity template with two roles: *Writer* and *Reader*.

The task-flow requirements for the *OnlineMeeting* activity is expressed below using path-expressions. The *OnlineMeeting* starts with either of the tasks: *TakeNotes* by the *Secretary*, *StartMeeting* by the *Coordinator*, or *NotesSharing* by the *Attendee*. The *StartMeeting* task has to be followed by either of any number of *ShareBoard* or *ReadBoard* tasks. The number of *NotesSharing* activity can be equal to the number of participants in the *Attendee* role. The *NotesSharing* activity can start with any number of *ReadNotes* or *WriteNotes* tasks and ends with *Submit*.

OnlineMeeting := (TakeNotes, StartMeeting;
(ShareBoard:*, ReadBoard:*), NotesSharing:#member(Attendee);)
NotesSharing := (ReadNotes:*,WriteNotes:*);Submit;

Several *role constraints*, *information flow*, and *access leakage* related requirements are specified for this collaboration environment.

RC1 *A member of the attendee role can never be a secretary.*

RC2 *The attendee who initiates a notes-sharing activity should be the only one who can join the writer role.*

IF3 *A secretary cannot access the content of the private shared notes unless provided by the owner of the notes.*

AL4 *A secretary should not be able to write on the BulletinBoard.*

AL5 *A participant of the writer role can only modify his/her notes before submission of his/her notes.*

A skeleton specification of the *OnlineMeeting* activity template is shown in Example 6. *Role admission* preconditions must be true when a user is admitted to a role. For example, the *Coordinator* role (line 05) has a role cardinality constraint, which requires that the member count for this role cannot exceed one. A *validation constraint* differs from an admission constraint as follow: (1) It can only be specified based on role membership related predicates; (2) It needs to be satisfied not only when a user is admitted to a role but also when a member initiates any role operations; and (3) It is also verified when any membership related query is invoked on the role. The role constraints `RC1` is expressed as a validation constraint in the *Secretary* role.

In Example 6 (line 20), using role reflection, all members of the *Coordinator* role are admitted to the *Attendee* role, subject to the *Attendee* role's admission constraints, in this case none. The template definition uses the *AssignedRoles* tag to specify which roles must be assigned at the time of activity instantiation, as specified for the *Writer* and *Reader* roles of the *NotesSharing* activity (line 34). When an attendee invokes the *ShareNotes* operation, an instance of the *NotesSharing* template is created, (line 29-30), and the participant creating the instance is dynamically assigned to the *Writer* role. Moreover, as assignment to the *Reader* role is not specified, the operation invoker is requested to assign members to the *Reader* role. Only member of the *Attendee* role can be assigned as specified by the precondition of the *Reader* role, (line 38).

Example 6 Collaborative Notes Taking

```
01 ActivityTemplate OnlineMeeting (AssignedRoles Coordinator,
02                                 Secretary, Attendee ) {
03   Role Coordinator{
04     AdmissionConstraints
```

```
05      #members(thisRole) < 1
06    Operation StartMeeting {
07      Precondition #(StartMeeting.start)=0
08      Action { board=new Object(BulletinBoard);}
09  }
10  Role Secretary {
11    ValidationConstraints
12      !member(thisUser, Attendee)
13    Operation ReadBoard {
14      Precondition #(StartMeeting.finish)=1
15      Action { data= board.read(); }}
16    Operation TakeNotes {
17      Precondition #(TakeNotes.start)=0
18      Action { notes=new Object (Notepad); Grant notes {write}}
19  }
20  Role Attendee (Reflect Coordinator){
21    ValidationConstraints
22      !member(thisUser, Secretary)
23    Operation ShareBoard {
24      Precondition #(StartMeeting.finish)=1
25      Action { data= board.read(); Grant board {write} }}
26    Operation ShareNotes {
27      Precondition #(ShareNotes.start(invoker = thisUser))=0
28      Action shared_notes=new Object (Notepad);
29        act=newActivity NotesSharing(
30                        shared_notes, Writer=thisUser)}
31  }
32  ActivityTemplate NotesSharing (Owner thisActivity.Creator,
33                       Object(Notepad notes)
34                       AssignedRoles ( Writer, Reader ) {
35    TerminationCondition #Submit.finish>0
36    Role Reader {
37      AdmissionConstraints
38        member(thisUser, parentActivity.Attendee)
39      Operation ReadNotes {
40        Action { Grant notes {read} }}
41    }
42    Role Writer () {
43      AdmissionConstraints
44        member(thisUser, parentActivity.Attendee)
45        ∧ member(thisUser, thisActivity.Creator)
46      Operation WriteNotes {
47        Action Grant notes {write} }
48      Operation Submit {
49        Action ChangeOwner(notes, Secretary) }
50    }
51 }}
```

In (line 14), a student in the *Secretary* role cannot execute the *TakeNotes* operation until the *Manager* has performed *StartMeeting* operation. On the other hand, the precondition for *ShareNotes* operation, (line 27), illustrates an intra-role coordination policy, which allows each member in the *Attendee* role to independently start a notes-sharing activity. As specified in (line 35), an *NotesSharing* activity terminates when the *Writer* performs the *Submit* operation. The *ShareNotes.finish* event is generated as soon as the activity *act* is instantiated; however, the *act.finish* event is generated when the *act* activity is terminated. The role constraint `RC2` is specified based on the *Creator* role in (line 45). Moreover, the owner of any *NoteSharing* activity instance is the creator of the instance, (line 33). In (line 49), the owner of the *notes* object is changed to the *Secretary* to satisfy requirements `IF3` and `AL5`.

7.4. Collaborative Document Authoring

Figure 7.2 introduces the overall model of our authoring example specified as a *DocumentAuthoring* activity template. The figure shows an instance of the activity template, named *document*. Authoring of an 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*.

The specification of the *DocumentAuthoring* activity template is presented in Example 6. In the *DocumentAuthoring* activity template, the nested *ChapterAuthoring* activity template 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. Figure 7.2 shows two instances of this template, named*chapter1* and *chapter2*. 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

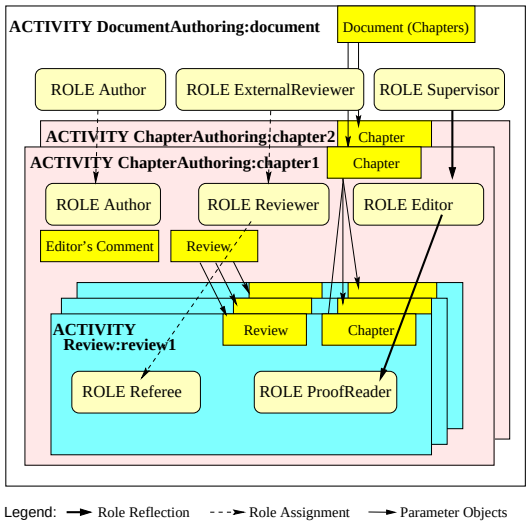

Figure 7.2. Collaborative Document Authoring Activities

represented as a task-flow diagram, shown in Figure 7.3. 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.

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

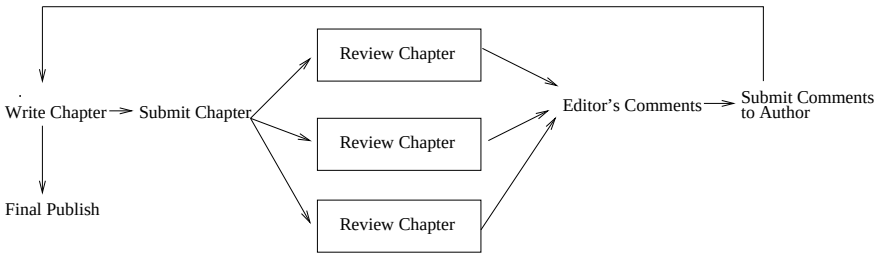

Figure 7.3. Task flow diagram of ChapterAuthoring Activity

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.

Example 6 Collaborative Document Authoring

```
Activity_Template DocumentAuthoring (Objects (Document doc),
                                     Assigned_RoleS Author, ExternalReviewer, Supervisor) {
  }
  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_Template ChapterAuthoring (Owner parentActivity.Supervisor,
                                    Object (DocumentAuthoring.Chapter chapter),
                                    Assigned_RoleS Author) {
  Termination_Condition #(Author.FinalPublish.finish) > 0
  Role Author {
    Admission_Constraints
      member(thisUser, parentActivity.Author)
    Operation WriteChapter{
      Precondition #(WriteChapter.start) − #(SubmitChapter.finish)=0
                   & #(SubmitChapter.start) − #(Editor.SubmitComment.finish)=0}
      Action Grant chapter {writeContent} }
    Operation SubmitChapter {
      Precondition #(Write.finish) − #(SubmitChapter.start)>0}
    Operation ReadComment {
      Precondition #(Editor.SubmitComment.finish)>0
      Action Grant 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 { Grant comment {writeComment} }}
     Operation SubmitComment {
      Precondition #(Author.SubmitChapter.finish) − #(SubmitComment.start) > 0
                   & #(WriteComment.finish) > 0 }
```

```
  }
Activity_Template Review(Owner parentActivity.Editor,
                         Object (DocumentAuthoring.Chapter chapter,
                                 DocumentAuthoring.Review review)) {
  Role Referee {
    Admission_Constraints
      member(thisUser, parentActivity.Reviewer)
      & thisActivity.creator=thisUser
      & #members(thisRole)<1
    Operation Read {
      Action Grant chapter {readContent}}
    Operation Write {
      Action Grant review {writeReview}}
    Operation Submit {
      Precondition #(Write.finish)>0}
  }
  Role ProofReader (Reflect parentActivity.Editor) {
    Activation_Constraints
      #(Referee.Submit.finish)>0
    Operation ReadReview {
      Action Grant review {readReview}}
  }
}
```

7.5. Evaluation of the Specification Model

In this section, we discuss based on our experience, the expressiveness of our specification model for security as well as coordination policies. Table 7.1 presents different types of security policies, as discussed in Chapter 2, and corresponding support by our specification model. In the following subsections, we discuss the security and coordination policies supported by the specification model. We also discuss why certain types of policies cannot be specified in the specification model.

7.5.1. Task based Security Policies

In the specification model, tasks are modeled as activities and operations. An activity is defined as a set of operations, and within an activity, these operations are grouped under roles. An operation performs actions including creating and accessing resources. Precondition can be specified to impose dependency among operations and activities. Based on these facilities, we can express task based security policies. The examples presented in thesis, specifically *Course* in Section 4.3, *Collaborative Notes*

Types of Security Requirements	Current Support
Task based Security Policies	Yes
Role based Separation-of-duties	Yes
Operational Separation-of-duties	Yes
History based Separation-of-duties	Yes
Confidentiality	Yes
Role Admission Control	Yes
Administrative RBAC	Yes
Identity based Separation-of-duties	Limited
Chinese Wall policy	Limited
Delegation	Limited
Privacy	Limited
Context Sensitive Security Policies	Limited
Obligation	No
Usage Control	No

Table 7.1
Security policies and thier support in the specification model

Taking in Section 7.3, and *Collaborative Document Authoring* in Section 7.4, uses tasks and task flow constraints as primary requirement during modeling corresponding specification.

7.5.2. Separation-of-Duties

In the specification model, based on admission, validation, and activation constraints "role based separation-of-duties" policies can be specified. Admission constraints are enforced during users' admission to roles enforcing static separation-of-duties policies among roles (see Section 4.4.9). Based on role validation and activation constraints that are enforced during users' activity in a role, dynamic separation-of-duties policies are specified (see Sections 4.4.8 and 4.4.11). "Operational separation-of-duties" are expressed using operation preconditions (see Section 4.4.10). Based on event and event count based predicates on the event history of a collaboration, "history based separation-of-duties" are expressed.

The specification model does not properly support "identity based Separation-of-duties". Example of an *identity-based separation of duty* requirement can be that users B and C cannot both be assigned concurrently to a role. This requirement can be expressed as shown below.

```
(thisUser≠B ∨ !member(C,thisRole)) ∧ (thisUser≠C ∨ !member(B,thisRole))
```

However, in the specification model, users are identified only with their roles. User identity, such as B and C cannot be specified in a specification, as shown above. A solution is to represent each user with a unique role. For example, two roles `B` and `C` can be exclusively assigned to users B and C, respectively to express the above requirement. The specification of this requirement is shown below.

```
(!member(thisUser,B) ∨ #(members(thisRole)\members(C))=#members(thisRole))
∧ (!member(thisUser,C) ∨ #(members(thisRole)\members(B))=#members(thisRole))
```

A specific user can be assigned to a role during instantiation of the top level activity to ensure exclusive roles. For proper support of "identity based separation-of-duties", a solution is to support exclusive roles for users in the specification model.

7.5.3. Chinese Wall Policy

The specification model is not developed from the security perspective of users tasks as oppose to classification of data. The collaboration specification model does not provide construct to specify policies on classes of objects. Based on roles, we can expresses Chinese Wall security policies, as presented in Section 2.2, by associating roles with "conflict of interest classes" and datasets with operations.

In the example of Chinese Wall security policy in Section 2.2, datasets for a financial institution are grouped into two conflict of interest classes: *Oil Company* and *Insurance Company*. The class *Oil Company* has datasets for multiple companies,

e.g., *A*, *B*, and *C* and the *Insurance Company* has datasets for companies *X*, *Y*, and *Z*. Once an agent of the financial institution accesses the dataset *A*, the agent is prohibited from accessing any other datasets of the class *Oil Company*; however he/she can access a dataset, either *X*, *Y*, or *Z*, of the class *Insurance Company*. By representing *Oil Company* as roles where each operation can only access a dataset among *A*, *B*, and *C*. By expressing an operational separation-of-duties, we can ensure a user can perform only one operation ensuring that he/she can access only one of the datasets.

7.5.4. Delegation

In the specification model, a role owner can assign users to the roles under his/her ownership. However, such member assignment has to follow role constraints, if any. Moreover, a role owner can remove a member from the role. In the model, during creation of an activity instance users in certain roles must be assigned by the creator of the activity instance. This also present a restricted form of delegation.

7.5.5. Privacy

A collaboration specification can be designed based information flow related requirements. We also take into account the trust on the administrative roles who may have access to information due to administrative privileges. Different trust facts can be expressed, from the viewpoint of the users based on their roles, as requirements to design collaboration environments. However, these policies resemble confidentiality as oppose to privacy, as individual users cannot express their privacy requirements. For example, the specification model does not support privacy agreement.

7.5.6. Context-Sensitive Security Policies

There is a wide range of context, such as time, users in a collaboration, physical location, coordination state of the cooperating users, proximity to devices, or any other application defined contextual information. There is an additional challenge in expressing context-sensitive security policies: modeling context so that they can be properly utilized for expressing context-dependent policies. In the specification model, we support modeling of activities, roles, operations, and exporting object state using application defined event in the coordination space. These including

the *date* function provide a wide range of context that we can utilize for expressing context-dependent security policies. However, we do not support other contextual information in a collaboration environments, such as physical locations of the users.

7.5.7. Administrative RBAC

A primary contribution of the specification model presented in this thesis is an administrative RBAC. We have developed based on the concepts of protection domain, security domain, and trust domain, an owner assignment model for roles, activities, and objects in a collaboration. Moreover, role admission and validation constraints enforce users admission and usage of roles in decentralized management of roles. These constraints not only enforce required credentials when a user joins a role but also revoke roles when specified validation constraints are not satisfied.

7.5.8. Obligation

Obligation policies guarantee the actions that will be performed when certain condition becomes true. These policies are also known as management policies as they pro-actively perform some tasks. These types of policies have real use for collaboration environments. For example, in the *WhiteBoard* sharing example, a policy may require automatic revocation of the board access after a predefined time. The specification model can be extended to support obligation policies by introducing the construct *Reaction*. *Reaction* can be substituted with an *Operation*; however the action of the reaction is triggered as soon as the precondition becomes true.

7.5.9. Usage Control

Access control policies are related to usage control that revoke access-right after a usage limit is reached. To support such policies, the specification model can be extended to query object state as part of the preconditions. As a precondition is only checked when an action is initiated, it cannot revoke permissions during an object-session initiated through an operation. For proper support for usage control, new construct needs to be introduced in the specification model. The conditions specified with this construct can be checked every time the specified object is accessed.

7.5.10. Coordination Policies

The specification model can specify policy related inter- and intra- role coordination requirements and structuring of activities. The specification model supports task based coordination as well as coordination on events that are generated based on objects state. However, this does not support fine grain coordination of shared view applications, such as browser buttons. These levels of coordination mechanisms are built into the applications. Interfaces can be exposed by the application for external control on the application artifacts.

7.6. Summary

In this chapter, we presented six example collaboration environments. The first four examples use the same *WhiteBoard* object but enforce different coordination polices. The *Audio Sharing* and the *Collaborative Notes Taking* present how security policies are specified. The *Collaborative Document Authoring* example presents how a structured document is supported by the specification model. The evaluation section presents the types of security policies that have limited or no support by the specification model. We also have discussed possible extension of the specification model to fully support security policies related to obligation, usage control, Chinese Wall security policy, and identity-based separation-of-duties.

Chapter 8

Conclusions

Web has become an integrated part of our life, enabling applications for interacting with users and services – from traditional electronic mail to recent web-based tax filing, distance learning, and a wide range of web based collaboration systems. On the other hand, the enabling open network infrastructure has raised security concerns, for every day applications, that were traditionally present in government or large scale commercial systems.

Social aspects for usage of these Internet-wide collaboration systems have raised demand for security services that were not prevalent earlier. For example, due to recent privacy concerns and laws in healthcare information systems, confidentiality has become a primary concern for many collaboration systems. Utilizing the Internet, these systems share data crossing their organizational and security boundaries. Enforcement of confidentiality policies is a challenge as an execution monitor to ensure secrecy cannot be complete (Volpano, 1999). To ensure secrecy, static analysis of a system, as utilized in this thesis, for confidentiality properties is an option.

Traditional security models relied on central trusted administrator as well as centralized logging facilities to resolve hard security problems, such as access-leakage and leakage of information. For distributed collaboration, where users from multiple domains with mutual distrust or lack of trust can interact, the security models need

to be revisited. Security policy specification needs to take into account distributed enforcement mechanisms that may not be trusted. Decentralized administration of security policies is a challenge for these systems.

Contemporary security policies express subjects' access on objects under specified conditions. Role based models gained popularity in security policy specification as they can express wide range of constraints, such as separation-of-duties. On the other hand, RBAC models need to address many challenges: modeling and engineering of roles, specification models for expressing constraints, and verification to ensure that the specified constraints are correct and consistent.

The work presented in this thesis is motivated by constructing secure distributed collaboration systems based on pre-generated specifications. Similar approaches are also present in Web-Services; however, Web-Services are concerned with orchestration of cross-organizational workflow processes. Our goal is to provide users with templates, which are designed based on different security and coordination requirements. Moreover, a policy-driven middleware constructs the distributed collaboration environment based on a selected template.

The primary contribution of this thesis is a methodology for specification and verification of security and coordination requirements in decentralized CSCW systems. The development of the methodology has the following research contributions:

- Development of a role based model for specifying security and coordination policies based on unique requirements present in distributed collaboration systems. This includes administrative security requirements in the presence of untrusted users due to different and independent security and trust domains.

- Development of finite-state model checking techniques for static verification of collaboration specifications. Based on the aspects of the requirements, an incremental verification methodology is developed with four verification models to handle problems related to state space explosion and inter dependency of the

requirements.

The second contribution of this thesis is the design of a policy driven middleware that constructs a distributed collaboration system based on a specification. The design is motivated by the several novel requirements, such as policy driven construction, selection of secure site for policy enforcement, decentralized trust, secure distributed event service and session management, decentralized coordination consistency management, and adaptation of the middleware with the changes in the specification model.

Appendix A

Syntax of the schema

ActivityTemplateDef ⟶ **ActivityTemplate** templateId [**Owner** roleId]
{ **Object** codebase objId} {**AssignedRoles** roleId}
[**TerminationCondition** *Condition*]
RoleDef {*RoleDef*} {*ActivityTemplateDef*}

RoleDef ⟶ **Role** roleId {**Reflect** roleId} [**Owner** roleId]
[**AdmissionConstraints** *Condition*]
[**ValidationConstraints** *RoleCondition*]
[**ActivationConstraints** *Condition*]
{*OperationDef*}

OperationDef ⟶ **Operation** opId [**PreCondition** *Condition*]
[**Action** actionDef]

ActionDef ⟶ {**Grant** *Permission*}
{*NewObjectDef*}
[*NewActivityDef*]
[**InvokeMethod** objId methodSignature methodParameter]
{**ChangeOwner** objId RoleId}
[**NotifyEvent** appDefinedEventId {*AttributeName=Attribute Value*}]

Permission ⟶ objId methodSignature

NewObjectDef ⟶ objId = **new Object** codebase

NewActivityDef ⟶ activityId = **new Activity** templateId {**PassedObject** objId}
{**MemberAssignment** roleId = userId {userId} }

Condition ⟶ *RoleCondition*
| *OperationCondition* | *TimeCondition*
| *Condition BinOp Condition*
| !*Condition*

RoleCondition ⟶ *#RoleMemberList Relation Count*
| member(*userId, roleId*)

RoleMemberList ⟶ members(roleId)
| *RoleMemberList SetOp RoleMemberList*

SetOp ⟶ ∩ | ∪ | \

TimeCondition ⟶ date *Relation String*

OperationCondition ⟶*EventCount Relation Count*
| *EventName*(*Index*)[*AttributeName Relation AttributeValue*]

EventCount ⟶ *#eventName* [*AttributeName Relation AttributeValue*]
| *EventCount IntegerOp EventCount*
| *EventCount IntegerOp Count*

EventName ⟶ EventId.start | EventId.finish

EventId ⟶ opId | activityId | appDefinedEventId

BinOp ⟶ ∧ | ∨

Relation ⟶ > | < | = | <= | >= | ≠

IntegerOp ⟶ + | - | `mod` | `div` | *

Count ⟶ 0 | 1 | ... | N

Index ⟶ *Count* | *EventCount* | `first` | `last`

AttributeName ⟶ invoker | date | *String*

AttributeValue ⟶ thisUser | *String*

String ⟶ { XML CDATA }

Appendix B

Schema for Activity Template in DTD

```
<?xml version="1.0" encoding="utf-8"?>
<!-- DTD for Activity -->

<!ELEMENT ACTIVITY_TEMPLATE (PASSED_OBJECT*, TERMINATION_CONDITION?,
                             ROLE+, ACTIVITY_TEMPLATE*)>
<!ATTLIST  ACTIVITY_TEMPLATE
 NAME CDATA #REQUIRED
 IDENTIFIER ID #REQUIRED
 OWNER IDREF #IMPLIED
 ASSIGNED_ROLES IDREFS #IMPLIED
>
<!ELEMENT PASSED_OBJECT EMPTY>
<!ATTLIST PASSED_OBJECT
  OBJECT_CONTEXT_REF IDREF #REQUIRED
  NAME CDATA #REQUIRED
  IDENTIFIER ID #REQUIRED
>
<!ELEMENT TERMINATION_CONDITION (DATE |IS_MEMBER| MEMBER_COUNT
```

```
                                |EVENT_CONDITION
                                |EVENT_INDEX_CONDITION
                                |AND | OR)>
<!ELEMENT DATE EMPTY>
<!ATTLIST DATE
  IDENTIFIER ID #REQUIRED
  RELATION (gt | lt | lte | gte | eq) "eq"
  MONTH (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) "Jan"
  DAY CDATA #REQUIRED
  YEAR CDATA #REQUIRED
  HOUR CDATA  #REQUIRED
  MIN CDATA  #REQUIRED
>
<!ELEMENT IS_MEMBER EMPTY>
<!ATTLIST IS_MEMBER
  TYPE (positive|negative) "positive"
  ROLE IDREF #REQUIRED
  PRINCIPAL CDATA #REQUIRED
>
<!ELEMENT MEMBER_COUNT ( MEMBERS | INTERSECT | UNION | DIFFERENCE) >
<!ATTLIST MEMBER_COUNT
  IDENTIFIER ID #REQUIRED
  RELATION (gt | lt | lte | gte | eq) "eq"
  VALUE NMTOKEN #REQUIRED
>
<!ELEMENT MEMBERS EMPTY >
<!ATTLIST MEMBERS
  ROLE IDREF #REQUIRED
>
<!ELEMENT INTERSECT (MEMBERS | INTERSECT | UNION | DIFFERENCE)+>
<!ELEMENT UNION (MEMBERS | INTERSECT | UNION | DIFFERENCE)+>
```

```
<!ELEMENT DIFFERENCE (MEMBERS | INTERSECT | UNION | DIFFERENCE)+>
<!ELEMENT EVENT_COUNT (EVENT_ATTRIBUTE*)>
<!ATTLIST EVENT_COUNT
  IDENTIFIER ID #REQUIRED
  EVENT_TYPE IDREF #IMPLIED
  SUFFIX (start|finish) "finish"
>
<!ELEMENT EVENT_ATTRIBUTE EMPTY>
<!ATTLIST EVENT_ATTRIBUTE
 NAME CDATA #REQUIRED
 RELATION (gt | lt | lte | gte | eq) "eq"
 VALUE CDATA #REQUIRED
>
<!ELEMENT EVENT_CONDITION (EVENT_COUNT | EVENT_COUNT_ADDITION
                           | EVENT_COUNT_SUBTRACTION
                           | EVENT_COUNT_MULTIPLICATION
                           | EVENT_COUNT_DIVISION)>
<!ATTLIST COMPLEX_EVENT_COUNT_CONDITION
  IDENTIFIER ID #REQUIRED
  TYPE (positive|negative) "positive"
  RELATION (gt | lt | lte | gte | eq) "eq"
  VALUE NMTOKEN #REQUIRED
>
<!ELEMENT EVENT_COUNT_ADDITION (EVENT_COUNT | EVENT_COUNT_ADDITION
                                | EVENT_COUNT_SUBTRACTION
                                | EVENT_COUNT_MULTIPLICATION
                                | EVENT_COUNT_DIVISION | CONSTANT)+>
<!ELEMENT EVENT_COUNT_SUBTRACTION (EVENT_COUNT | EVENT_COUNT_ADDITION
                                   | EVENT_COUNT_SUBTRACTION
                                   | EVENT_COUNT_MULTIPLICATION
                                   | EVENT_COUNT_DIVISION
```

```
                                    | CONSTANT)+>
<!ELEMENT EVENT_COUNT_MULTIPLICATION (EVENT_COUNT
                                    | EVENT_COUNT_ADDITION
                                    | EVENT_COUNT_SUBTRACTION
                                    | EVENT_COUNT_MULTIPLICATION
                                    | EVENT_COUNT_DIVISION
                                    | CONSTANT)+>
<!ELEMENT EVENT_COUNT_DIVISION (EVENT_COUNT | EVENT_COUNT_ADDITION
                                | EVENT_COUNT_SUBTRACTION
                                | EVENT_COUNT_MULTIPLICATION
                                | EVENT_COUNT_DIVISION | CONSTANT)+>
<!ELEMENT CONSTANT EMPTY>
<!ATTLIST CONSTANT
    VALUE NMTOKEN #REQUIRED
>
<!ELEMENT EVENT_INDEX_CONDITION ( EVENT_ATTRIBUTE+)>
<!ATTLIST EVENT_INDEX
  IDENTIFIER ID #REQUIRED
  EVENT_TYPE IDREF #IMPLIED
  INDEX CDATA #REQUIRED
  SUFFIX (start|finish) "finish"
>
<!ELEMENT AND (DATE | IS_MEMBER| MEMBER_COUNT| EVENT_CONDITION
              ! EVENT_INDEX_CONDITION | AND|OR)+>
<!ATTLIST AND
  TYPE (positive|negative) "positive"
>
<!ELEMENT OR (DATE | IS_MEMBER| MEMBER_COUNT| EVENT_CONDITION
             | EVENT_INDEX_CONDITION| AND| OR)+>
<!ATTLIST OR
  TYPE (positive|negative) "positive"
```

```
>
<!ELEMENT METHOD_SIGNATURE EMPTY>
<!ATTLIST METHOD_SIGNATURE
  NAME NMTOKEN #REQUIRED
  RETURN_TYPE IDREF #IMPLIED
  PARAM_TYPE CDATA #IMPLIED
>
<!ELEMENT PERMISSION (METHOD_SIGNATURE)>
<!ELEMENT ROLE (ADMISSION_CONSTRAINTS?, VALIDATION_CONSTRAINTS?,
                ACTIVATION_CONSTRAINTS?, OPERATION*)>
<!ATTLIST ROLE
  NAME CDATA #REQUIRED
  IDENTIFIER ID #REQUIRED
  OWNER IDREF #IMPLIED
  ASSIGN IDREFS #IMPLIED
  ROLE_INTERFACE NMTOKEN #IMPLIED
>
<!ELEMENT ADMISSION_CONSTRAINTS (DATE | IS_MEMBER| MEMBER_COUNT
                                | EVENT_CONDITION
                                | EVENT_INDEX_CONDITION
                                 | AND| OR)+>
<!ELEMENT VALIDATION_CONSTRAINTS (DATE | IS_MEMBER| MEMBER_COUNT
                                 | EVENT_CONDITION
                                 | EVENT_INDEX_CONDITION
                                 ! AND| OR)+>
<!ELEMENT ACTIVATION_CONSTRAINTS (DATE | IS_MEMBER| MEMBER_COUNT
                                 | EVENT_CONDITION
                                 | EVENT_INDEX_CONDITION
                                 | AND| OR)+>
<!ELEMENT OPERATION (PRECONDITION?, ACTION?)>
<!ATTLIST OPERATION
```

```
  NAME NMTOKEN #REQUIRED
  IDENTIFIER ID #REQUIRED
>
<!ELEMENT PRECONDITION (DATE | IS_MEMBER| MEMBER_COUNT| EVENT_CONDITION
                      | EVENT_INDEX_CONDITION| AND| OR)>
<!ELEMENT ACTION (NEW_OBJECT*, GRANT_PERMISSION?, NEW_ACTIVITY?,
                  METHOD_INVOCATION?, NOTIFY_EVENT? )>
<!ELEMENT NEW_OBJECT EMPTY>
<!ATTLIST NEW_OBJECT
  NAME CDATA #REQUIRED
  IDENTIFIER ID #REQUIRED
  CODEBASE CDATA #REQUIRED
  OWNER IDREF #IMPLIED
>
<!ELEMENT GRANT_PERMISSION EMPTY>
<!ATTLIST GRANT_PERMISSION
  CODEBASE CDATA #REQUIRED
  OBJECT_REF IDREF #REQUIRED
  INVOKED_METHODS CDATA #REQUIRED
>
<!ELEMENT NEW_ACTIVITY (ROLE_ASSIGNMENT*)>
<!ATTLIST NEW_ACTIVITY
  NAME CDATA #REQUIRED
  IDENTIFIER ID #REQUIRED
  ACTIVITY_TEMPLATE_REF IDREF #REQUIRED
  OBJECT_REFS IDREFS #IMPLIED
  OWNER IDREF #IMPLIED
>
<!ELEMENT ROLE_ASSIGNMENT (PRINCIPAL+)>
<!ATTLIST ROLE_ASSIGNMENT
  ROLE IDREF #REQUIRED
```

```
>
<!ELEMENT PRINCIPAL EMPTY>
<!ATTLIST PRINCIPAL
  URN CDATA #REQUIRED
>
<!ELEMENT METHOD_INVOCATION (METHOD_SIGNATURE, PARAMETER*)>
<!ATTLIST METHOD_INVOCATION
  OBJECT_REF IDREF #REQUIRED
>
<!ELEMENT PARAMETER EMPTY>
<!ATTLIST PARAMETER
  VALUE IDREF #REQUIRED
>
<!ELEMENT NOTIFY_EVENT (NOTIFY_EVENT_ATTRIBUTE*)>
<!ATTLIST NOTIFY_EVENT
  IDENTIFIER ID #REQUIRED
  NAME CDATA #REQUIRED
>
<!ELEMENT NOTIFY_EVENT_ATTRIBUTE (METHOD_INVOCATION | ATTRIBUTE_VALUE) >
<!ATTLIST NOTIFY_EVENT_ATTRIBUTE
 NAME CDATA #REQUIRED
>
<!ELEMENT ATTRIBUTE_VALUE EMPTY >
<!ATTLIST ATTRIBUTE_VALUE
 VALUE CDATA #IMPLIED
>
```

Appendix C

Specification of Course activity template

```
ActivityTemplate Course (AssignedRoles Assistant, Instructor, Student) {
  Role Assistant{
    AdmissionConstraints
      #members(thisRole) < 1
      ∧ !member(thisUser, Student)
      ∧ !member(thisUser, Instructor)
      ∧ #members(Instructor) > 0
    Operation AccessBoard {
      Precondition #(InitiateBoard.start)=1
      Action {Grant board.read; board.write }}
  }
  Role Instructor {
    AdmissionConstraints
      !member(thisUser, Student)
      ∧ !member(thisUser, Assistant)
    Operation InitiateBoard {
      Precondition #(InitiateBoard.start)=0
      Action { board=new Object(BulletinBoard);
               Grant board.read; board.write }}
    Operation StartExamination {
      Action exam=new Activity Examination(Examiner={})}
  }
  Role Student {
    AdmissionConstraints
      !member(thisUser,Instructor)
      ∧ !member(thisUser, Assistant)
    Operation AccessBoard {
```

```
    Precondition #(InitiateBoard.start)=1
    Action {Grant board.read; board.write }}
}

ActivityTemplate Examination AssignedRoles Examiner Owner Instructor {
  TerminationCondition #(exam_session.finish)=#members(Examinee)
  Role Examiner {
    AdmissionConstraints
            member(thisUser, parentActivity.Instructor)
    Operation SetPaper {
            Precondition #(SetPaper.start)=0
            Action { exam=new Object(ExamPaper); Grant exam.setQuestions(data) }}
  }
  Role Approver (Owner Adm2) {
    ValidationConstraints
            !member(thisUser, parentActivity.Assistant) ∧ !member(thisUser, parentActivity.Student)
    Operation ApprovePaper {
            Precondition #(SetPaper.finish)=1 ∧ #(SetPaper.finish(invoker=thisUser))=0 }}
  }
  Role Examinee (Reflect parentActivity.Student) {
    Operation StartExam {
            Precondition #(ApprovePaper.finish)=1 ∧ #(StartExam.start(invoker=thisUser))=0
            Action session=new Activity ExamSession( exam, Candidate=thisUser)}
  }
  Role Grader(Reflect parentActivity.Assistant, parentActivity.Instructor){
    ValidationConstraints !member(thisUser, Approver)
  }

  ActivityTemplate ExamSession( ExamPaper exam, AssignedRoles Candidate)
    Owner Creator {
    TerminationCondition #(Checker.Grade.finish)>0
    Role Candidate {
      AdmissionConstraints
        member(thisUser, parentActivity.Examinee)
        ∧ member(thisUser, thisActivity.Creator)
        ∧ #members(thisRole)<1
      ActivationConstraints
        date>DATE(May, 10, 2003, 9:00) ∧date<DATE(May, 10, 2003, 11:00)
      Operation OpenExam{
        Precondition #(OpenExam.start)=0
        Action { ans=new OBJECT(AnswerBook);Grant exam.readPaper() }
      Operation Write {
        Precondition #(OpenExam.finish)>0
        Action Grant ans.writeAnswer(data) }
      Operation Submit {
        Precondition #(Write.finish)>0
        Action ChangeOwner(ans, Checker) }
```

```
      }
      Role Checker {
        AdmissionConstraints
          #members(thisRole)<1 ∧ member(thisUser, parentActivity.Grader)
        Operation Grade {
          Precondition #(Candidate.Submit.finish)=1
          Action Grant ans.setGrade(data)}
      }
    }
  }
}
```

Appendix D

Specification Editor

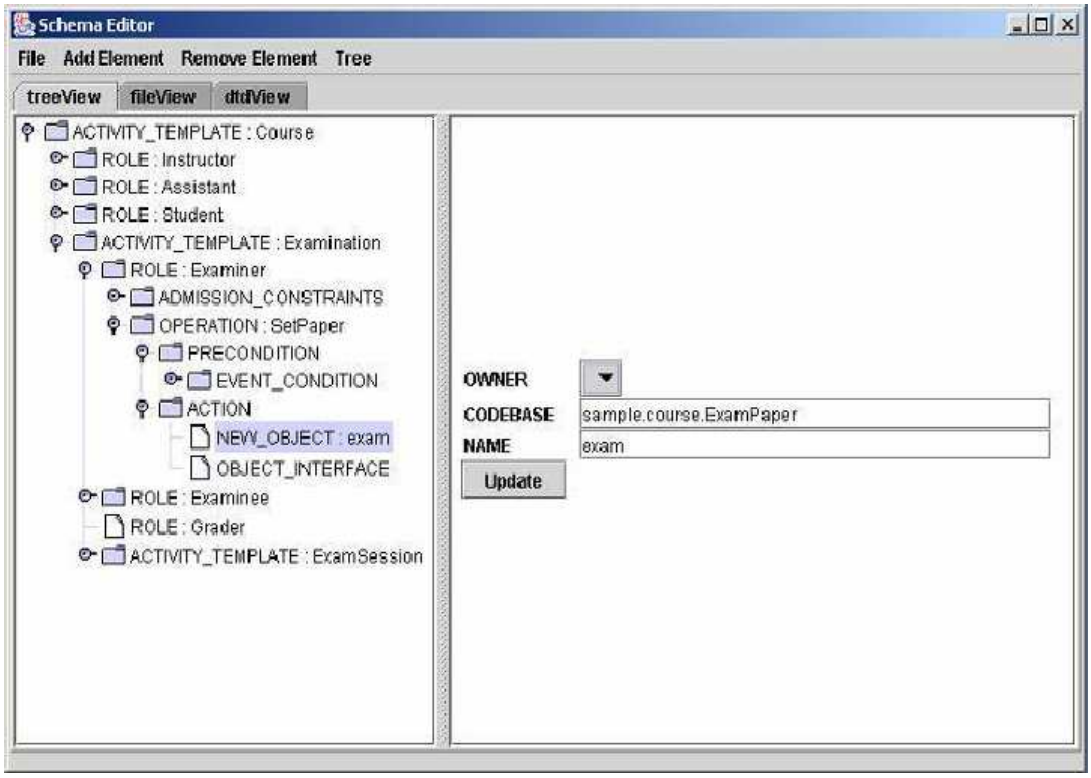

Figure D.1. Editor for collaboration specification

Bibliography

Abdul-Rahman, A., Hailes, S., 1998. A distributed trust model. In: Proceedings of the Workshop on New Security Paradigms Workshop. ACM, New York, pp. 48 – 60.

Ackerman, M. S., 2000. The Intellectual Challenge of CSCW: The gap between social requirements and technical feasibility. Human-Computer Interaction 15, 179–203.

Ahmed, T., Tripathi, A., December 2003a. Specification and Verification of Security Requirements in Decentralized CSCW Systems. Tech. rep., Department of Computer Science, University of Minnesota, available at http://www.cs.umn.edu/Ajanta/publications.html.

Ahmed, T., Tripathi, A. R., June 2003b. Static Verification of Security Requirements in Role Based CSCW Systems. In: Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003). ACM, New York, pp. 196–203.

Ahn, G.-J., Sandhu, R., November 2000. Role-based authorization constraints specification. ACM Transactions on Information and System Security 3 (4), 207 – 226.

Anderson, R. J., May 1996. A security policy model for clinical information systems. In: IEEE Symposium on Security and Privacy. pp. 30 – 43.

Andrews, G. R., Reitman, R. P., January 1980. An Axiomatic Approach to Information Flow in Programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (1), 56–76.

Azzedin, F., Maheswaran, M., April 2003. Trust modeling for peer-to-peer based computing systems. In: Parallel and Distributed Processing Symposium. pp. 99 –108.

Bacon, J., Moody, K., Yao, W., November 2002. A Model of OASIS Role-Based Access Control and its Support for Active Security. ACM Transactions on Information and System Security 5 (4), 492 – 540.

Baldwin, R., 1990. Naming and grouping privileges to simplify security management in large databases. In: IEEE Computer Society Symposium on Research in Security and Privacy. pp. 116 –132.

Banavar, G., Doddapaneni, S., Miller, K., Mukherjee, B., 1998. Rapidly Building Synchronous Collaborative Applications By Direct Manipulation. In: Proceedings of CSCW'98. pp. 139–148.

Bardram, J., 1998. Designing for the Dynamics of Cooperative Work Activities. In: Proceedings of CSCW'98. pp. 89–98.

Basin, D., Doser, J., Lodderstedt, T., June 2003. Model driven security for process-oriented systems. In: Proceedings of the eighth ACM symposium on Access control models and technologies . pp. 100–109.

Bell, D. E., La Padula, L., 1975. Secure Computer System: Unified Exposition and Multics Interpretation. Tech. rep., ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA.

Bertino, E., Ferrari, E., February 1999. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. ACM Transactions on Information and System Security 2 (1), 65 – 104.

Bertino, E., Ferrari, E., Atluri, V., 1997. A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. In: ACM Workshop on Role-based Access Control. pp. 1–12.

Bevier, W. R., Young, W. D., June 1994. A state-based approach to noninterference. In: Proceedings of 7th Computer Security Foundations Workshop. pp. 11–21.

Biddle, B. J., 1966. Role theory: concepts and research. New York, Wiley, edited by Bruce J. Biddle and Edwin J. Thomas.

Bishop, M., 2000. Computer Security: Art and Science. Addison Wesley Professional.

Blaze, M., Feigenbaum, J., Lacy, J., 1996. Decentralized Trust Management. In: Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 164–173.

Brewer, D., Nash, M., 1989. The Chinese Wall Security Policy. In: 1989 IEEE Symposium on Security and Privacy. pp. 206 –214.

Brothers, L., Sembugamoorthy, V., Muller, M., 1990. ICICLE: groupware for code inspection. In: Proceedings of the conference on Computer-supported cooperative work. pp. 169 – 181.

Bullock, A., Benford, S., November 1999. An access control framework for multi-user collaborative environments. In: Proceedings of the international ACM SIGGROUP conference on Supporting group work. pp. 140 – 149.

Campbell, R. H., Habermann, A. N., April 1974. The Specification of Process Synchronization by Path Expressions. In: Operating Systems, International Symposium, Rocquencourt. Lecture Notes in Computer Science vol.16, Springer Verlag.

Carla Simone, Monica Divitini, K. S., August 1995. A notation for malleable and interoperable coordination mechanisms for CSCW systems. In: Proceedings of Conference on Organizational Computing Systems. pp. 44 – 54.

Casey, T. J., Vinter, S., Weber, D., Varadarajan, R., Rosenthal, D., April 1988. A secure distributed operating system. In: IEEE Symposium on Security and Privacy. pp. 27 –38.

Christine M. Neuwirth, David S. Kaufer, R. C., Morris, J. H., 1990. Issues in the design of computer support for co-authoring and commenting. In: Proceedings of the conference on Computer-supported cooperative work. pp. 183 – 195.

Clark, D. D., Wilson, D. R., 1987. A Comparison of Commercial and Military Computer Security Policies. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 184–194.

Cohen, A. L., Cash, D., Muller, M. J., December 2000. Designing to support adversarial collaboration. In: Proceeding on the ACM Conference on Computer supported cooperative work. pp. 31 – 39.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, Zheng, H., 2000. Bandera: Extracting Finite-State Models from Java Source Code. In: Proceedings of International Conference on Software Engineering. pp. 439 – 448.

Corts, M., Mishra, P., November 1996. DCWPL: a programming language for describing collaborative work. In: Proceedings of CSCW'96. pp. 21 – 29.

D. Wodtke, J. Weissenfels, G. W., Kotz-Dittrich, A., 1996. Mentor Project: Steps Towards Enterprise-Wide Workflow Management. In: International Conference on Data Engineering.

Damianou, N., Dulay, N., Lupu, E., Sloman, M., 2001. The Ponder Policy Specification Language. In: POLICY. pp. 18–38.

Danielson, T., Pankoke-Babatz, U., 1988. The Amigo Activity Model. In: R.Speth (Ed.), Research into Networks and Distributed Applications. Elsevier Science Publishers B.V., North Holland, pp. 227–241.

Demurjian, S., Ting, T., Thuraisingham, B., Summer 1993. User-role based security for collaborative computing environments. Multimedia Review 4 (2), 40–47.

Denning, D. E., August 1993. A new paradigm for trusted systems. In: Proceedings on the 1992-1993 Workshop on New Security Paradigms. ACM, New York, pp. 36 – 41.

Denning, D. E., Denning, P. J., July 1977. Certification of programs for secure information flow. Communications of the ACM 20 (7), 504–513.

Denning, D. E. R., 1982. Cryptography and Data Security. Addison-Wesley.

DePaoli, F., Tisato, F., 1993. Development of a collaborative application in CSDL. In: Proceedings the 13th International Conference on Distributed Computing Systems. pp. 210 –217.

DePaoli, F., Tisato, F., 1994a. Cooperative systems configuration in CSDL. In: Proceedings of the 14th International Conference on Distributed Computing Systems. pp. 304 –311.

DePaoli, F., Tisato, F., August 1994b. CSDL: a language for cooperative systems design. IEEE Transactions on Software Engineering 20 (8), 606 –616.

Department of Health and Human Services, 2004. Standards for Privacy of Individually Identifiable Health Information. Available at URL http://aspe.hhs.gov/admnsimp/final/pvcguide1.htm.

Dewan, P., Shen, H., 1992. Access Control for Collaborative Environments. In: Proceedings of CSCW'92. pp. 51–58.

Dewan, P., Shen, H., March 1998a. Controlling access in multiuser interfaces. ACM Transaction Computer-Human Interaction 5 (1), 34 – 62.

Dewan, P., Shen, H., 1998b. Flexible Meta Access-Control for Collaborative Applications. In: Proceedings of CSCW'98. pp. 247–256.

Dobry, R., Schanken, M., Dec 1994. Security concerns for distributed systems. In: Proceedings of Computer Security Applications Conference. pp. 12 –20.

Dollimore, J., Wilbur, S., September 1989. Experiences in building a configurable CSCW system. In: Proceedings 1st European Conference on CSCW. pp. 215–225.

Dourish, P., 1999. Software infrastructures. In: Beaudouin-Lafon, M. (Ed.), Computer Supported Co-operative Work. Vol. 7 of Trends in Software. John Wiley & Sons Ltd., pp. 195–219.

Du Li, Z. W., Muntz, R. R., February 1999. "Got COCA?" A new perspective in building electronic meeting systems. In: Proceedings of the International Joint Conference on Work Activities Aoordination and Collaboration. pp. 89 – 98.

Ellis, C., Keddara, K., August 2000. ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. Computer Supported Cooperative Work (CSCW)The Journal of Collaborative Computing 9 (3/4), 293–333.

Ellis, C. A., Gibbs, S. J., 1989. Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD international conference on Management of data. pp. 399 – 407.

Ellis, C. A., Gibbs, S. J., Rein, G., January 1991. Groupware: some issues and experiences. Communication of ACM 34 (1), 39 – 58.

Epstein, P., Sandhu, R., December 2001. Engineering of Role/Permission Assignments. In: 17th Annual Computer Security Applications Conference .

Eshuis, R., Wieringa, R., 2002. Verification Support for Workflow Design with UML Activity Graphs. In: Proceedings of International Conference on Software Engineering. ACM, New York, pp. 166 – 176.

Fine, T., 1996. Defining Noninterference in the Temporal Logic of Actions. In: IEEE Symposium on Security and Privacy. pp. 12–23.

Focardi, R., Gorrieri, R., 1997. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. IEEE Transactions on Software Engineering 23 (9), 550–571.

Fuchs, N., Sept. 1992. Specifications are (preferably) executable. Software Engineering Journal 7 (5), 323 – 334.

Furuta, R., Stotts, P. D., October 1994. Interpreted collaboration protocols and their use in groupware prototyping. In: Proceedings of the conference on Computer supported cooperative work. pp. 121 – 131.

Gelernter, D., Carriero, N., February 1992. Coordination languages and their significance. Communications of the ACM 35 (2), 96–107.

Giuri, L., Iglio, P., November 1997. Role templates for content-based access control. In: Proceedings of the Second ACM Workshop on Role-Based Access Control. pp. 153 – 159.

Gligor, V., Gavrila, S., Ferraiolo, D., 1998. On the formal definition of separation-of-duty policies and their composition. In: Proceedings IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 172–183.

Graubart, R., Oct. 1989. On the need for a Third Form of Access Control. In: Proceedings of the 12th National Computer Security Conference. pp. 296–304.

Gravell, A., Henderson, P., March 1996. Executing formal specifications need not be harmful . Software Engineering Journal 11 (2), 104 – 110.

Greenberg, S., 1991. Personalizable Groupware: Accomodating Individual Roles and Group Differences. In: Proceedings of the European Conference on Computer-Supported Cooperative Work(ECSCW). pp. 17–31.

Greif, I., Sarin, S., 1987. Data sharing in group work. ACM Transactions on Information Systems 5 (2), 187–211.

Grudin, J., 1994. Groupware and Social Dynamics: Eight Challenges for Developers. Communications of the ACM 37 (1), 92–105.

Grudin, J., Poltrock, S. E., 1997. Computer-Supported Cooperative Work and Groupware. Advances in Computers 45, 269–320.

Gunter, C., Gunter, E., Jackson, M., Zave, P., May/June 2000. A reference model for requirements and specifications. IEEE Software 17 (3), 37–43.

Guruduth Banavar, Sri Doddapaneni, K. M., Mukherjee, B., 98. Rapidly Building Synchronous Collaborative Applications By Direct Manipulation. In: CSCW 98. pp. 139–148.

Guzdial, M., Rick, J., Kerimbaev, B., December 2000. Recognizing and supporting roles in CSCW. In: ACM Conference on Computer Supported Cooperative Work. pp. 261 – 268.

H. Gajewska, J. Kistler, M. M., Redell, D., 1994. Argo: a system for distributed collaboration. In: Proceedings of the second ACM international conference on Multimedia. pp. 433 –440.

Harney, H., Colegrove, A., McDaniel, P., February 2001. Principles of Policy in Secure Groups. In: Proceedings of Network and Distributed Systems Security.

Harrison, M. A., Ruzzo, W. L., Ullman, J. D., August 1976. Protection in Operating Systems. Communications of the ACM 19 (8), 461 – 471.

Heimdahl, M. P., Thompson, J. M., October 2000. Specification based prototyping of control systems. In: 19th IEEE Digital Avionics Systems Conference. Philadelphia.

Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F., Wilner, W., June 1994. The Rendezvous architecture and language for constructing multiuser applications. ACM Transactions on Computer-Human Interaction 1 (2), 81 – 125.

HOL, 2003. The HOL Theorem Prover. Available at URL http://www.dcs.gla.ac.uk/ tfm/fmt/hol.html.

Hollingsworth, D., 2004. The Workflow Reference Model: 10 Years On. Tech. rep., Workflow Management Coalition, available at http://www.wfmc.org/information/info.htm.

Holzmann, G. J., 2003. SPIN Model Checker, The: Primer and Reference Manual. Addison Wesley Professional.

Huang, W.-K., Atluri, V., 1999. SecureFlow: A Secure Web-enabled Workflow Management System. In: ACM Workshop on Role-based Access Control. ACM, New York, pp. 83 – 94.

Hudson, S. E., Smith, I., November 1996. Techniques for addressing fundamental privacy and disruption tradeoffs in awareness support systems. In: Proceedings of ACM conference on computer supported cooperative work . pp. 248 – 257.

Jaeger, T., Prakash, A., 1996. Requirements of role-based access control for collaborative systems. In: Proceedings of the first ACM Workshop on Role-based access control. pp. 253 – 264.

Jaeger, T., Tidswell, J. E., May 2001. Practical Safety in Flexible Access Control Models. ACM Transactions on Information and System Security 4 (2), 158 – 190.

Jajodia, S., Kudo, M., Subrahmanian, V. S., 2001. Provisional authorization. In: In Proc. Ecommerce Security and Privacy. Kluwer Academic Publishers, pp. 133–159.

Jajodia, S., Samarati, P., Subrahmanian, V. S., 1997. A Logical Language for Expressing Authorizations. In: IEEE Symposium on Security and Privacy. pp. 31 –42.

Janssen, W., Mateescu, R., Mauw, S., Springintveld, J., 1998. Verifying Business Processes using Spin. In: Proceedings of 4th International SPIN Workshop.

Johnson, P., Dec. 1992. Supporting exploratory CSCW with the EGRET framework. In: Proceedings of the 1992 ACM conference on Computer-supported cooperative work. pp. 298–305.

Jonathan Trevor, Tom Rodden, G. B., September 1993. COLA: A Lightweight Platform for CSCW. In: Proceedings of ECSCW'93. pp. 5–30.

Jsang, A., 1996. The right type of trust for distributed systems. In: Proceedings of the UCLA Conference on New Security Paradigms Workshops. pp. 119 – 131.

Kammer, P. J., Bolcer, G. A., Taylor, R. N., Hitomi, A. S., Bergman, M., August 2000. Techniques for supporting dynamic and adaptive workflow. Computer Supported Cooperative Work 9 (3/4), 269 – 292.

Kaplan, S. M., Tolone, W. J., Bogia, D. P., Bignoli, C., 1992. Flexible, active support for collaborative work with ConversationBuilder. In: Proceedings on Computer-Supported Cooperative Work. pp. 378 – 385.

Karnik, N., Tripathi, A., January 2001. Security in the Ajanta Mobile Agent Programming System. Software Practice and Experience , 301–329.

Kawatsura, Y., June 2003. RFC 3538 - Secure Electronic Transaction (SET) Supplement for the v1.0 Internet Open Trading Protocol (IOTP). Available at URL http://www.faqs.org/rfcs/rfc3538.html.

Kiczales, G., Jan 1996. Beyond the black box: open implementation. IEEE Software 13 (1), 8, 10 –11.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An overview of AspectJ. In: Knudsen, J. L. (Ed.), ECOOP 2001 Object-Oriented Programming, Lecture Notes in Computer Science. Vol. 2072. pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., Irwin, J., 1997. Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (Eds.), ECOOP'97 Object-Oriented Programming, Lecture Notes in Computer Science. Vol. 1241. pp. 220–242.

Kling, R., December 1991. Cooperation, Coordination and Control in Computer-Supported Work. Communications of the ACM 34 (12), 83 – 88.

Koch, M., Mancini, L. V., Parisi-Presicce, F., August 2002. A graph-based formalism for RBAC. ACM Transactions on Information and System Security 5 (3), 332 – 365.

Kotonya, G., Sommerville, I., 1998. Requirements engineering : processes and techniques. John-Wiley & Sons, Chichester,New York.

Kumar, R., 2002. Protocols for Secure Interactions in Distributed Collaboration Systems, MS Plan B Report. Tech. rep., University of Minnesota, available at http://www.cs.umn.edu/Ajanta/publications.html.

Kyng, M., 1991. Designing for Cooperation: Cooperating in Design. Communications of the ACM 34 (12), 65–73.

Lai, R., Jirachiefpattana, A., 1998. Communication protocol specification and verification. Kluwer Academic,Boston.

Lampson, B. W., October 1973. A note on the confinement problem. Communications of the ACM 16 (10).

Landay, J. A., Davis, R. C., 1999. Making sharing pervasive: Ubiquitous computing for shared note taking. IBM Systems Journal 38 (4), 531–550.

Lauwers, J. C., Lantz, K. A., 1990. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In: Conference on Empowering People: Human Factors in Computing System. pp. 303 – 311.

Leland, M., Fish, R., Kraut, R., 1988. Collaborative Document Production using Quilt. In: Proceedings of CSCW'88. pp. 206–215.

Li, D., Muntz, R., 1998. COCA: Collaborative Objects Coordination Architecture. In: Proceedings of CSCW'98. pp. 179–188.

Li, D., Muntz, R. R., November 1999. Runtime dynamics in collaborative systems. In: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work. pp. 336 – 345.

Li, W., Wang, W., Marsic, I., November 1999. Collaboration transparency in the DISCIPLE framework. In: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work. pp. 326 – 335.

Lunt, T., Denning, D., Schell, R., Heckman, M., Shockley, W., June 1990. The SeaView security model. IEEE Transactions on Software Engineering 16 (6), 593 –607.

Lupu, E. C., Sloman, M., 1997. Reconciling Role-Based Management and Role-Based Access Control. In: ACM Workshop on Role-based Access Control. ACM, New York, pp. 135–141.

Maggi, P., Sisto, R., 2002. Using SPIN to Verify Security Protocols. In: Proceedings of 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318. pp. 187–204.

Malone, T. W., Crowston, K., March 1994. The Interdisciplinary Study of Coordination. ACM Computing Surveys (CSUR) 26 (1), 87 – 119.

Malone, T. W., Lai, K.-Y., Fry, C., April 1995. Experiments with Oval: a radically tailorable tool for cooperative work . ACM Transactions on Information Systems (TOIS) 13 (2), 177 – 205.

Martinelli, F., 1998. Partial Model Checking and Theorem Proving for Ensuring Security Properties. In: Proceedings of 11th IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, Los Alamitos, CA, pp. 44 –52.

McLean, J., 1994. Security Models. In: Marciniak, J. (Ed.), Encyclopedia of Software Engineering. John Wiley & Sons.

Millen, J. K., 1981. Information Flow Analysis of Formal Specifications. In: IEEE Symposium on Security and Privacy. pp. 3 – 8.

Minsky, N., Ungureanu, V., September 1997. Regulated Coordination in Open Distributed Systems. In: Second International Conference on Coordination Languages and Models.

Mohammed, I., Dilts, D. M., 1994. Design for dynamic user-role-based security. Computers & Security 13 (8), 661–671.

Myers, A. C., Liskov, B., 2000. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology 9 (4), 410–442.

Natalie S. Glance, D. S. P., Pareschi, R., 1996. Generalized process structure grammars GPSG for flexible representations of work. In: Proceedings of the ACM conference on on Computer supported cooperative work. pp. 180 – 189.

Needham, R. M., Schroeder, M. D., December 1978. Using Encryption for Authentication in Large Networks of Computers. Commun. ACM 21 (12), 993–999.

Nyanchama, M., Osborn, S., February 1999. The Role Graph Model and Conflict of Interest. ACM Transaction on Information System Security 2 (1), 3–33.

OASIS Security Services TC , August 2003. Security Assertion Markup Language (SAML) v1.1. Available at URL http://www.oasis-open.org/specs.

Oravec, J. A., 1996. Virtual individuals, virtual groups : human dimensions of groupware and computer networking. Cambridge; New York: Cambridge University Press.

Ørbæk, P., Palsberg, J., Nov. 1997. Trust in the λ-calculus. Journal of Functional Programming 7 (6), 557–591.

Orlikowski, W. J., December 1992. Learning from Notes: Organizational Issues in Groupware Implementation. In: Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work. pp. 362 – 369.

Osborn, S. L., 2002. Information Flow Analysis of an RBAC System. In: ACM Symposium on Access Control Models and Technologies. ACM, New York, pp. 163 – 168.

Ossher, H., Tarr, P., October 2001. Using multidimensional separation of concerns to (re)shape evolving software. Communications of the ACM 44 (10), 43–50.

Park, J., Sandhu, R., February 2004. The UCONABC usage control model. ACM Transactions on Information and System Security (TISSEC) 7 (1).

Parnas, D. L., December 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12).

Reiter, M., Gong, L., 1995. Securing Causal Relationships in Distributed Systems. he Computer Journa 38 (8), 633–642.

Rekers, J., Sprinkhuizen-Kuyper, I., June 1993. A LOTOS specification of a CSCW tool. In: Proceedings of Design of Computer Supported Cooperative Work and Groupware Systems.

Roberts, P., Verjus, J.-P., 1977. Towards Autonomous Descriptions of Synchronization Modules. In: Proceedings of IFIP Congress. North-Holland, Amsterdam, pp. 981–986.

Roseman, M., Greenber, S., 1993. Building flexible groupware through open protocols. In: Proceedings of the Conference on Organizational Computing Systems. pp. 279 – 288.

Roseman, M., Greenberg, S., March 1996. Building real-time groupware with GroupKit, a groupware toolkit. ACM Transactions on Computer-Human Interaction 3 (1), 66 – 106.

Ryan, P., McLean, J., Millen, J., Gligor, V., 2001. Non-interference, who needs it? In: 14th IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, Los Alamitos, CA, pp. 237 –238.

Sandhu, R., 1992. The Typed Access Matrix Model. In: IEEE Computer Society Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 122 –136.

Sandhu, R., 1998. Role activation hierarchies. In: Proceedings of the third ACM workshop on Role-based access control. pp. 33 – 40.

Sandhu, R., Bhamidipati, V., Munawer, Q., February 1999. The ARBAC97 model for role-based administration of roles. ACM Transactions on Information and System Security 2 (1), 105 – 135.

Sandhu, R., Coyne, E., Feinstein, H., Youman, C., February 1996. Role-Based Access Control Models. IEEE Computer 29 (2), 38–47.

Sandhu, R., Ferraiolo, D., Kuhn, R., July 2000. The NIST model for role-based access control: towards a unified standard. In: Proceedings of the Fifth ACM Workshop on Role-based Access Control. ACM, New York, pp. 47–63.

Sandhu, R. S., December 1988. Transaction control expressions for separation of duties. In: Fourth Annual Computer Security Application Conference. pp. 282–286.

Schmidt, K., Simone, C., 1996. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. Computer Supported Cooperative Work 5 (2/3), 155–200.

Schneider, S., 1996. Security properties and CSP. In: IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 174 –187.

Sikkel, K., September 1997. A Group-based Authorization Model for Cooperative Systems. In: European Conference on Computer Supported Cooperative Work (ECSCW'97). pp. 345–360.

Simon, R., Zurko, M., 1997. Separation of duty in role-based environments. In: 10th Computer Security Foundations Workshop. IEEE Computer Society Press, Los Alamitos, CA, pp. 183 –194.

Simonet, V., 2002. Fine-grained Information Flow Analysis for a lambda-calculus with Sum Types. In: Proceedings of. 15th IEEE Computer Security Foundations Workshop. pp. 209 –223.

Sluizer, S., Cashman, P. M., 1985. XCP: an experimental tool for managing cooperative activity. In: Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science. pp. 251 – 258.

Smith, G., Rodden, T., 1993. Access as a means of configuring cooperative interfaces. In: Proceedings of the conference on Organizational computing systems. pp. 289 – 298.

Smith, R. B., Hixon, R., Horan, B., November 1998. Supporting flexible roles in a shared space. In: Proceedings of the ACM conference on Computer supported cooperative work. pp. 197–206.

Snyder, L., March 1981. Formal Models of Capability-Based Protection Systems. IEEE Transactions on Computers C-30 (3), 172 – 181.

Stafford, T. F., June 2003. E-Services. Communications of the ACM 46 (6), 27–28.

Suchman, L., 1994. Do Categories Have Politics? The language/action perspective reconsidered. Computer-Supported Collaborative Work: The Journal of Collaborative Computing 2 (3), 177–190.

Tavani, H. T., Moor, J. H., March 2001. Privacy protection, control of information, and privacy-enhancing technologies . ACM SIGCAS Computers and Society 31 (1), 6 –11.

Thomas, R. K., 1997. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In: ACM Workshop on Role-based Access Control. ACM, New York, pp. 13 – 19.

Tidswell, J. E., Jaeger, T., July 2000. Integrated constraints and inheritance in DTAC. In: ACM Workshop on Role-based Access Control. ACM, New York, pp. 93 – 102.

Ting, T., 1988. A User-Role Based Data Security Approach. Elsevier, editor C.E. Landwehr.

Trevor, J., Rodden, T., Mariani, J., 1994. The use of adapters to support cooperative sharing. In: Proceedings of the Conference on Computer Supported Cooperative Work. pp. 219 – 230.

Tripathi, A., Ahmed, T., Kumar, R., April 2003. Specification of Secure Distributed Collaboration Systems. In: IEEE International Symposium on Autonomous Distributed Systems. IEEE Computer Society Press, Los Alamitos, CA, pp. 149–156.

Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., Vora, M. K., Pathak, M., May 2002. Design of the Ajanta System for Mobile Agent Programming. Journal of Systems and Software 62 (2), 123–140.

Vardi, M. Y., Wolper, P., 1986. An automata-theoretic approach to automatic program verification. In: Proceedings First IEEE Symp. on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos, CA, pp. 322–331.

Volpano, D. M., 1999. Safety versus Secrecy. In: Static Analysis Symposium. Springer-Verlag, pp. 303–311.

Volpano, D. M., Smith, G., 2000. Verifying Secrets and Relative Secrecy. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, pp. 268–276.

W3C, April 2002a. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification, W3C Recommendation. Available at URL http://www.w3c.org/TR/P3P/.

W3C, 2002b. Web Services Activity. Available at URL http://www.w3.org/2002/ws/.

William J. Tolone, S. M. K., Fitzpatrick, G., August 1995. Specifying dynamic support for collaborative work within WORLDS. In: Proceedings of conference on Organizational computing systems COOCS'95. pp. 55 – 65.

Winograd, T., 1994. Categories, Disciplines, and Social Coordination. Computer-Supported Collaborative Work: The Journal of Collaborative Computing 2 (3), 191–197.

Workflow Management Coalition, 1999. Interface 1 - Process Definition Interchange V 1.1 Final. http://www.wfmc.org.

Yahalom, R., Klein, B., Beth, T., 1993. Trust relationships in secure systems-a distributed authentication perspective. In: IEEE Computer Society Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 150 –164.

Zakinthinos, A., Lee, E., 1997. A General Theory of Security Properties. In: IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA, pp. 94 –102.

Zurko, M. E., Simon, R. T., 1996. User-centered security. In: Proceedings of the UCLA Conference on New Security Paradigms Workshops. pp. 27 – 33.