

# Technical Report

Department of Computer Science  
and Engineering  
University of Minnesota  
4-192 EECS Building  
200 Union Street SE  
Minneapolis, MN 55455-0159 USA

TR 05-003

Specification and Verification of Security Requirements in  
Decentralized CSCW Systems

Tanvir Ahmed and Anand Tripathi

February 25, 2005



# Specification and Verification of Security Requirements in Decentralized CSCW Systems <sup>1</sup>

Tanvir Ahmed and Anand R. Tripathi  
Department of Computer Science  
University of Minnesota, Minneapolis MN 55455

## Abstract

In this paper, we present a specification model and a verification methodology for security policies in distributed CSCW systems. To express security and coordination requirements in decentralized CSCW systems, a role-based specification model is developed. We show how dynamic security requirements in collaboration environments are expressed in this specification model. Given global security requirements, verification of a specification ensures correctness and consistency of the specification. The goal of our methodology is to ensure that sensitive security requirements cannot be violated in decentralized management of a collaboration involving multiple security domains where all the participating users in the collaboration may not be trusted. We have utilized finite-state based model checking for static verification of security requirements. Several verification models are developed to check security properties, such as task-flow constraints, information flow or confidentiality, and assignment of administrative privileges.

## 1 Introduction

The examples of CSCW (Computer Supported Cooperative Work) systems include online conferencing, product design and development, authoring of documents, workflow in an office environment, and collaboration among different organizations. With the advent of Internet, recent CSCW systems are decentralized and span across organizations and security domains. Distributed collaboration environments, often termed as *virtual organizations*, are formed by ad hoc integration involving multiple users, trust domains, and peer groups with mutual distrust. Other than management and access control of distributed shared resources, distributed collaboration systems have to ensure proper coordination of users.

Security policies are concerned with subjects' access to resources under specified conditions. Policies related to security attributes – confidentiality, integrity, and resource access – in a CSCW system needs 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, thus representing the fact that coordination constraints are often weaved with access control concerns in collaboration systems. In collaboration systems, role-based models for coordination and security policy specification have been used for their ability to specify policies without knowing the users who will participate at runtime [GS87]. In role-based access control (RBAC) models, a role represents a set of privileges [SFK00]. A user assigned to a role acquires those privileges. Role-based security policies in collaboration and workflow systems have been found to be quite natural as participants perform a set of well-defined tasks pertaining to their expertise and responsibilities in the organization [DTT93].

Another challenge in specifying security policies for distributed CSCW systems is the expression of administrative level security requirements. In Internet-wide collaborations, users and shared objects are inherently distributed. Collaboration systems need decentralized management as no single

---

<sup>1</sup>This work was supported by National Science Foundation grant 0082215 and 0411961.

organization, site, or user may be trusted with all management aspects of a system. Without a proper or “trusted” assignment of users in *administrative roles* [SBM99], members 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 correctly enforce the part of the policy he 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 not enforcing the policies under their control or by omitting or falsifying events to manipulate the causal dependency of policies managed at a host.

The objective of the verification process is to ensure that the specified constraints do not violate any required coordination and security requirements, 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 access rights can be leaked to unauthorized users.

The work presented in this paper is developed as part of our broader research goal of realizing distributed CSCW systems from their specifications using a middleware, which we have implemented. The primary contribution of this paper is twofold:

1. Development of a role-based specification model for coordination and security policies, including administrative policies, in distributed collaboration systems.
2. Development of a verification methodology using model checking to ensure that a given specification satisfies the desired coordination and security requirements.

A focus of the verification process is on a methodology to ensure that sensitive security requirements cannot be violated by untrusted participants in distributed management of a collaboration system. 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 verification methodology, the problem is related to identifying the roles that cannot be trusted to enforce policies related to specific security requirements and determining *safe* assignments of administrative roles.

As part of the verification methodology, we present how finite-state techniques, such as model checking, can be utilized for correctness verification of a role-based specification. During the design phase of a collaboration, an existing model checking tool, SPIN [Hol03], is utilized to verify security requirements.

The following section discusses requirements for secure collaboration systems. Our role-based specification model for distributed collaboration is described in Section 3. Section 4 presents the security properties that need to be verified. Our verification methodology is presented in Section 5. Sections 6 and 7 present the related work and the conclusions, respectively.

## 2 Requirements for Secure Collaboration

Here we identify several important security and coordination requirements that have guided the development of our role-based model for specifying collaboration systems.

### 2.1 Role Admission Constraints

In role-based security models, a user acquires access rights by being a member of a role. Hence, to control the user's acquisition of access rights, a role-based model needs to address constraints on users' acquiring a role. These constraints can be based on events that must happen before a user could 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. RBAC models are centralized and only address user assignment to roles. In distributed systems, these role admission related constraints need to support the functionality of acquiring and revoking role memberships [BMY02].

### 2.2 Inter- and Intra- Role Coordination

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. However, when multiple users are allowed to be present simultaneously in a role, they may need to coordinate their actions, which is termed as *intra-role coordination*. 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.

### 2.3 Hierarchical Structuring of Activities

In an organization, many activities exist and new activities are constantly created. 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 work activities into tasks/subtasks and planing how such tasks will be performed.

### 2.4 Separation-of-Duties

*Separation-of-duties* requires that no single user has all the privileges to perform a sensitive task. The requirement of separation-of-duties also arises due to the need of avoiding conflict-of-interest situations [CW87]. The ability to express various forms of separation-of-duties is a primary motivation behind RBAC models [FCK95]. In RBAC models, various separation-of-duties constraints have been formalized [SZ97, GGF98].

## 2.5 Dynamic Access Control Policies

In a collaboration environment, the privileges assigned to a member in a role may change with time due to the actions executed by other participants. For example, in a course examination activity, a security requirement can be that students can only view the question after the examiner has released it and only during the specified time period of the exam-session activity. Sometimes permissions may change due to the user's own actions, such as making a final agreement on a document. RBAC and most of the existing MAC (Mandatory Access Control) and DAC (Discretionary Access Control) style security policies are static, i.e., they do not depend on time or any dynamically changing conditions. In collaboration environments, different permissions may need to be assigned to roles based on various contexts and events.

Several types of separation-of-duties constraints and history-based access control conditions also fall into the category of dynamic access control policies. In the context of role-based access control, dynamic access control policies have to address role cardinality-based requirements, e.g., a maximum or minimum number of members that must be present in a role for a member to acquire the role privileges.

## 2.6 Confidentiality

Confidentiality requirements express information-flow constraints. A collaboration system may need to hide the identity or other user-specific data of one participant from another. In such cases, the presence of a participant may only be visible through his/her role or a pseudonym in a role but not by name. It may be required to hide the identities of the members of a role from other roles. Consider a course examination activity, which has two roles: the *Candidate* takes the exam and the *Grader* grades the answer book. A confidentiality requirement can be that the graders do not know the identities of the *Candidate* role's members. As it is argued that a *reference monitor* to ensure secrecy cannot be complete [McL94] due to covert channels, static analysis for confidentiality properties are preferred.

## 2.7 Meta-Level Administrative Security Policies

In decentralized execution environment, participants from domains with mutual distrust need to manage the activities and shared resources in a collaboration. Policies related to role membership management, object access, task creation, task coordination, and other administrative aspects of managing collaborative activities require to be enforced in security domains managed by entities that 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 [BFL96]. Second, trust is defined as an entity's "trustworthiness", which is used to derive recommendation or reputation [ARH98]. 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. We term such an entity as an *administrative role*. If an administrative role is trusted, its members are trusted not to violate polices under their control. Meta policies need to be specified on administrative roles. Examples of meta policies include policies specifying who can be present in these administrative roles.

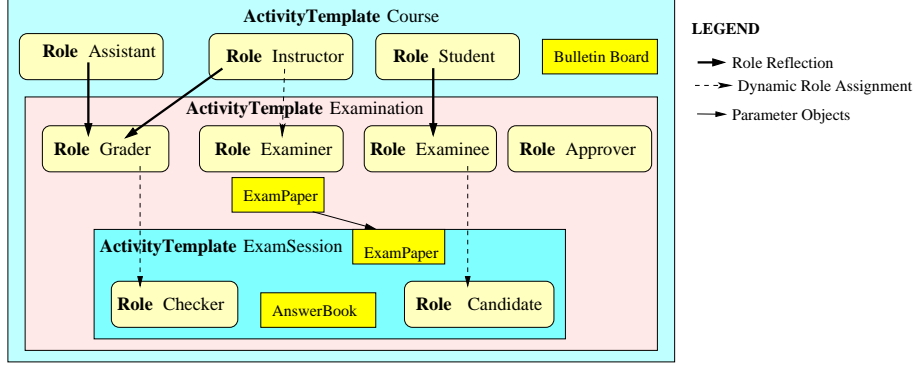


Figure 1: Role member assignment and object passing in hierarchical structuring of activities

### 3 A Role Based Specification Model for Decentralized Collaboration

In our collaboration model, an *activity* defines how a group of users cooperate toward some common objectives by sharing a set of 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. An activity can also span across multiple organizations.

#### 3.1 Activity Template

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 can be structured hierarchically, consisting of other nested activity templates. Within an activity, only immediately nested activity templates can be instantiated. Within an activity, only the A new activity instance is called a *child* activity, which is instantiated in the scope of the *parent* activity. Roles and objects are created within the scope of an activity instance.

Using Figure 1, we discuss three main concepts of the specification model: (1) hierarchical structuring of activities, (2) scope rules for objects and roles, and (3) assignment of role members and passing of objects as parameters to nested activities. In Figure 1, an activity template *Course* is presented that has three roles – *Instructor*, *Assistant*, and *Student* – with disjoint members. In the *Course*, a nested *Examination* activity template is defined with four roles: *Grader*, *Examiner*, *Examinee*, and *Approver*. An instance of the *Course* activity template can be *chemistry*. Within the *chemistry* activity instance, there can be more than one *Examination* activity instances, such as *midterm* and *final\_exam*. In an examination activity, each examinee takes an exam by instantiating the nested *ExamSession* activity template. An exam-session activity contains the roles *Candidate* and *Checker*. In the following sections, we discuss scope rules for roles and objects, role member assignment, and passing of objects as parameters to activities.

#### 3.2 Role

In our model, a role defines a set of *operations*. An operation represents *permissions* to perform some role specific tasks. A role member may exercise any or none of the permissions acquired as

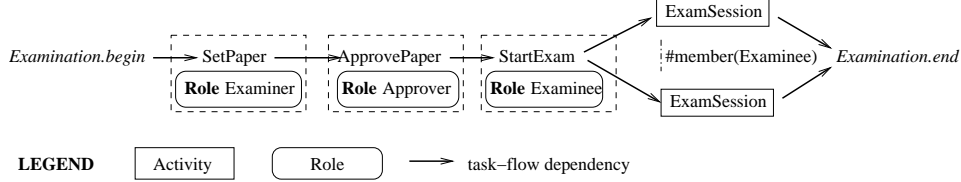


Figure 2: Task flow requirements in an *Examination* activity

part of an operation. An operation can create an object, invoke object methods, and instantiate a nested activity. A role operation can have a precondition that must be satisfied when the operation is invoked.

Execution of an operation generates *events*. Operation precondition can depend on the *event history*, i.e., all the previously occurred events, within an activity instance. Hence, a role operation can be coordinated with other operations defined only within the same activity instance.

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

- *Role admission constraints* must be satisfied when a user is admitted to a role.
- *Role validation constraints* must be satisfied not only during admission but throughout a user’s membership in the role. If these constraints are not satisfied, a user’s membership to the role is revoked.
- *Role activation constraints* must be satisfied for performing any role operations.

In this model, participants in roles from outer ancestor activities can join or be assigned to a role in a child activity. The specification model supports assignments of members in roles from an ancestor activity to roles in a child activity. There are two mechanisms that support such assignment of role members. The first mechanism is *static role assignment* or *role reflection*. In role reflection, all members of specified roles in the ancestor 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 role in the ancestor activity) also implies removal from the role in the child activity. In Figure 1, 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 second mechanism is *dynamic role assignment* in which role admission and validation constraints specify the members that can be admitted to a role at runtime. *Dynamic role assignment* can have an additional requirement in which users to be admitted in a role are specified at the time of activity instantiation. In Figure 1, the *Instructor* initiates an *Examination* activity by assigning members to the *Examiner* role. Each examinee creates an *ExamSession* activity instance to take the exam by assigning himself to the *Candidate* role. On the other hand, a member of the *Grader* role can join the *Checker* role after an exam-session instance is created.

Within an *Examination* activity, there are tasks that are performed by role members. For example a member of the *Examiner* sets the exam-paper, the *Approver* approves it, and the *Examinee* takes exam. These tasks are represented as operations and activities in the specification model as illustrated in Figure 2. The arrow in the figure shows the dependency among these operations and activities. For example, the approver can approve an exam-paper after the examiner sets the paper, an examinee can start his exam-session only after the *ApprovePaper* operation, and an examination terminates when exam-sessions of all the examinees terminate.



---

```

ActivityTemplateDef → ActivityTemplate templateId [Owner roleId]
                    {Object codebase objId} {AssignedRoles roleId}
                    [TerminationCondition Condition]
                    RoleDef {RoleDef} {ActivityTemplateDef}

```

---

Figure 3: Syntax for activity template definition

### 3.3 Object

Objects are defined and are created within an activity scope. Objects can be passed as parameters to a nested activity during instantiation of the activity. Only the specified types of objects can be created as part of a role operation. Only the roles within an activity can have access to an object created within the activity.

In Figure 1, within a *Course* activity, a member of the *Instructor* role can create a *BulletinBoard* object. Only members of the *Instructor*, *Assistant*, and *Student* roles within this activity, if permitted, can access the *BulletinBoard*. The *BulletinBoard* cannot be accessed by roles in any child activity instances, if not passed as a parameter. In Figure 1, in an *Examination* activity, a reference to the *ExamPaper* object is passed as a parameter to nested *ExamSession* activities. 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 in other exam-session activities.

### 3.4 Owner 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 [Den93]. During initiating secure sessions, trust is assumed among interacting distributed hosts to correctly perform specific tasks. Based on these specific tasks, trust is classified, such as providing identification of another entity, maintaining secret, and not interfering with other entities' sessions [YKB93].

In the distributed collaboration model, we classify trust for proper management of an entity – activity instance, role, and object. Based on the aspects of the policies that are enforced on an entity, trust is classified representing *trust classes*, such as, managing role membership, enforcing coordination policies, and enforcing access control policies. For each entity, we have defined a meta role *Owner* who is trusted to manage the entity-specific policies. 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 has unrestricted access to the object, and the owner of an activity can terminate the activity.

### 3.5 Activity Template Specification

An activity template is a collaboration specification. Based on our collaboration model, we have devised an XML schema for collaboration specifications. Here, rather than using XML, we use a notation that is simple to read and conceptually easy to follow. In Figure 3, 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. Based on the schema presented in Figure 3, an activity template must have an id and at least one role

```

1  ActivityTemplate Course AssignedRoles Assistant, Instructor, Student {
2    Role Assistant{....}
3    Role Instructor {....}
4    Role Student {....}
5    ActivityTemplate Examination Owner Instructor AssignedRoles Examiner {
6      Role Examiner { .... }
7      Role Approver { .... }
8      Role Examinee Reflect parentActivity.Student { .... }
9      Role Grader Reflect parentActivity.Assistant, parentActivity.Instructor{....}
10     ActivityTemplate ExamSession Owner Creator Object ExamPaper exam
11       AssignedRoles Candidate {
12         Role Candidate { .... }
13         Role Checker { .... }
14   } } }

```

Figure 4: Skeleton specification of *Course* activity template

defined and can have other nested templates. The declaration of an activity template can have the owner assignment, formal parameters for objects with ids and Java classes that uniquely identify the types of the passed objects, and a termination condition. Moreover, the declaration may list some of its roles that must be assigned members during creating an activity instance.

In Figure 4, a partial specification of the *Course* activity template of Figure 1 is presented. The complete specification of the nested *Examination* and *ExamSession* activity templates are presented in Figure 8 and Figure 10, respectively. The specification in Figure 4 provides an example of nested declaration of activity templates. The *Examination* and *ExamSession* activity templates in Figure 8 and 10 provide example specifications of various coordination and security requirements.

In the specification model, an activity, object, or role encapsulated in the scope of an activity can be referenced by a fully qualified name. In Figure 4, the *Instructor* role in the *Course* activity template can be referenced as *Course.Instructor*. A name can be assigned for newly created activities or objects.

In the specification, the user currently executing an operation is identified by *thisUser*. Within an activity, one can refer to its current instance using the pseudo variable *thisActivity* and its parent activity instance using *parentActivity*. In Figure 4 (line 9), the *Grader* role refers to the *Assistant* role of its parent activity using *parentActivity.Assistant*. Within a role, one can refer to the role by *thisRole*.

### 3.6 Condition Specification

There are mainly two types of conditions in the specification model: conditions based on role membership functions and conditions based on event history-based functions, as defined in Figure 5. For time-based condition specification, we also support a function *time* that returns current time.

#### 3.6.1 Role Membership Functions

A boolean function `member(thisUser, roleId)` checks if the currently executing 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)`.

---

```

Condition → RoleCondition | OperationCondition | TimeCondition
           | Condition LogicOp Condition | !Condition
RoleCondition → #RoleMemberList Relation Count | member(userId, roleId)
RoleMemberList → members(roleId) | RoleMemberList SetOp RoleMemberList
TimeCondition → time Relation String
SetOp → ∩ | ∪ | \
Relation → > | < | = | <= | >= | ≠
LogicOp → ∧ | ∨
String → { XML CDATA }

```

---

Figure 5: Syntax for condition definition: time and role membership based predicates

---

```

OperationCondition → EventCount Relation Count
                   | EventName '['Index']'.'AttributeName Relation AttributeValue
EventCount → #eventName [AttributeName Relation AttributeValue]
           | EventCount IntegerOp EventCount | EventCount IntegerOp Count
EventName → opId.start | opId.finish | activityId.start | activityId.finish
Index → Count | EventCount | first | last
AttributeName → invoker | time | String
AttributeValue → thisUser | String
IntegerOp → + | - | mod | div | *
Count → Integer

```

---

Figure 6: Syntax for condition definition: event based predicates

### 3.6.2 Event Based Predicates

There are two classes of events. First, *activity events* are related to instantiation and termination of activities. Second, *operation events* represent execution of role operations. These events are specified with corresponding names, i.e., `opId` and `activityId`. Associated with each event class, there are two types of events – `start` and `finish` – representing the initiation and termination of an operation or an activity. Events are implicitly generated by the system, based on the model in [RV77]. As shown in Figure 6, `EventName` represents four types of events – `opId.start`, `opId.finish`, `activityId.start`, and `activityId.finish`.

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

For each activity and operation events, there are two predefined attribute-names: *invoker* and *time*. An event type can be derived by filtering an event-list based on some predicate on the event's attributes. The expression `opId.start(invoker=thisUser)` defines a filter based on the operation invoker's identity. Using the count operator, the expression  `#(opId.start(invoker=thisUser))` counts the number of times the currently executing user has previously invoked this operation.

When an operation is requested and the operation's precondition is satisfied, the operation's *start* event is generated. 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 at the end of the operation's actions.

Conditions are specified as predicates based on events. These predicates are expressed in two

---

```

RoleDef → Role roleId [Owner roleId] {Reflect roleId}
          [AdmissionConstraints Condition] [ValidationConstraints RoleCondition]
          [ActivationConstraints Condition] {OperationDef}

```

---

Figure 7: Syntax for role definition

ways as shown in Figure 6. For example,

1. The predicate, `#op1.start-#op2.start=0`, is true when the operations `op1` and `op2` have started equal number of times.
2. The predicate, `opId.start[last].invoker≠thisUser`, is true if the invoker, who has initiated the last `opId` operation, is not same as the current invoker.

### 3.7 Role Specification

A role specification, as shown in Figure 7, primarily contains specification for the operations within the role and three types of role constraints: role admission, validation, and activation constraints. Moreover, a role declaration includes a role name and can have an owner and the ids of other roles whose members are *reflected* into this role. In the following subsections, we present different security requirements that can be expressed by role constraints and operation preconditions.

#### 3.7.1 Role Admission on Activity Creation

As discussed in Section 3.2, the specification model supports two mechanisms for assignment of role members. First, using the *Reflect* tag, members of the roles in ancestor activities are statically assigned to roles in a *child* activity. In Figure 4, partial specification of the roles in Figure 1 is presented. In Figure 4 (lines 8 and 9), members are assigned to the *Examinee* and *Grader* roles through *role reflection*.

Second, the template specification uses the *AssignedRoles* tag to specify the roles for which some member must be assigned at the time of activity instantiation. In Figure 4 (line 1), members of the *Instructor*, *Assistant*, and *Student* roles must be assigned when instantiating a *Course* activity. Similarly, in Figure 4 (lines 5 and 11), members of the *Examiner* and *Candidate* roles must be assigned at the time of instantiating an *Examination* and an *ExamSession* activity, respectively.

#### 3.7.2 Role Admission Constraints

These must be true for a user to be admitted to a role. These constraints control a user’s admission to the role to meet various security requirements including *static separation-of-duties* requiring that two given roles should never be assigned to the same user. The following admission constraints for the *Assistant* role in the *Course* activity are selected to illustrate various aspects of security requirements that can be expressed using role admission constraints.

- A minimum role cardinality constraint requires that the member count for this role cannot exceed one.

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

```

1 ActivityTemplate Examination Owner Instructor AssignedRoles Examiner {
2   TerminationCondition #exam_session.finish=#members(Examinee)
3   Role Examiner {
4     AdmissionConstraints member(thisUser, parentActivity.Instructor)
5     Operation SetPaper {
6       Precondition #(SetPaper.start)=0
7       Action { exam=new Object(ExamPaper); Grant exam setQuestions }}}
8   Role Approver Owner Adm2 {
9     ValidationConstraints
10    !member(thisUser, parentActivity.Assistant) ^ !member(thisUser, parentActivity.Student)
11    Operation ApprovePaper {
12      Precondition #(SetPaper.finish)=1 ^ #(SetPaper.finish(invoker=thisUser))=0 }}}
13   Role Examinee Reflect parentActivity.Student {
14     Operation StartExam {
15       Precondition #ApprovePaper.finish=1 ^ #StartExam.start(invoker=thisUser)=0
16       Action { session=new Activity ExamSession PassedObject exam
17              MemberAssignment Candidate=thisUser}
18   Role Grader Reflect parentActivity.Assistant, parentActivity.Instructor{
19     ValidationConstraints !member(thisUser, Approver) }
20 }

```

Figure 8: Specification of *Examination* activity template

- A role admission pre-requisite constraint requires that a user is admitted to this role only when at least one member is present in the *Instructor* role.

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

- A *static separation-of-duties* requires that the same person cannot be assigned to both the *Student* and *Assistant* roles.

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

To ensure this *static separation-of-duties*, the following constraint is also specified in the *Student* role of the *Course* activity.

```
!member(thisUser, Assistant)
```

### 3.7.3 Role Validation Constraints

A user's membership to a role is revoked when the role validation constraints are not satisfied. *Previous qualifications* that must be current during role operation invocation are specified as validation constraints. Also, *dynamic separation-of-duties* constraints, such as two given roles cannot be concurrently assigned to the same person, are specified as part of role validation constraints.

Figure 8 presents the specification of the roles in the *Examination* activity template with role constraints. In Figure 8 (lines 10 and 19), the *Approver* and the *Grader* roles have validation constraints. The validation constraints for the *Approver* role specifies that a user's membership to the *Approver* role is revoked if the user becomes a member of the *Assistant* or the *Student* role. The validation constraint for the *Grader* role specifies that when a member the *Grader* role becomes a member of the *Approver* role, his/her membership to the *Grader* role is revoked.

---

```

OperationDef → Operation opId [Precondition Condition] [Action actionDef]
ActionDef → { Grant Permission } { NewObjectDef } [ NewActivityDef ]
           [ InvokeMethod objId methodSignature methodParameter ]
           { ChangeOwner objId RoleId }
Permission → objId methodSignature
NewObjectDef → objId = new Object codebase
NewActivityDef → activityId = new Activity templateId { PassedObject objId }
                { MemberAssignment roleId = userId {userId } }

```

---

Figure 9: Syntax for role operation definition

### 3.7.4 Operation Specification

As presented in Figure 9, an operation specification includes a name, and may include a *precondition* and an *action*. The precondition must be true when the operation is invoked.

The preconditions associated with operations allow one to specify coordination constraints as well as various dynamic security requirements. Dynamic security requirements in this model are specified as condition-based access control constraints. Preconditions can also express *history-based separation-of-duties* that imposes predefined order on the actions performed by different roles.

An operation action can create a new object or a nested activity, invoke a method on an object, or change ownership of an object. 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. Roles can create only predefined types of objects, specified with a codebase, as defined with *NewObjectDef* in Figure 9.

Figure 8 presents the *Examination* activity template with the operations and their corresponding dependency requirements as expressed in Figure 2. In Figure 8 (lines 5-7), also presented below, the *Examiner* role in the *Examination* can perform the *SetPaper* operation only once as specified by the operation precondition. This operation results in creation of an *exam* object of type *ExamPaper* and granting the operation invoker the *setQuestions* privilege on the object.

```

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

```

Preconditions also enable us to specify coordination constraints, for both *inter-role* and *intra-role* coordination. For example, in Figure 8 (line 15), a student in the *Examinee* role cannot execute the *StartExam* operation until the *Approver* has approved the exam paper. This represents an *inter-role* coordination constraint. Moreover, the precondition for this operation only allows each member in the *Examinee* role to independently start an exam session. This illustrates an intra-role coordination policy, specifically the *independent participation model* for the members in the *Examinee* role.

In Figure 8 (line 6), the precondition of the *SetPaper* operation in the *Examiner* specifies that only one of the role members can execute the *SetPaper* operation. This illustrates the *cooperative participation model* of *intra-role* coordination policies.

An *operational separation-of-duties* constraint, i.e., no single participant can perform all the operations related to a business transaction, is specified for the *Approver* role in Figure 8 (lines 11-12). 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 has prepared.

```

1 ActivityTemplate ExamSession Owner Creator Object ExamPaper exam AssignedRoles Candidate{
2   TerminationCondition #Checker.Grade.finish>0
3   Role Candidate {
4     AdmissionConstraints member(thisUser, parentActivity.Examinee)
5                       ^ member(thisUser, thisActivity.Creator)
6                       ^ #members(thisRole)<1
7     ActivationConstraints time > DATE(May, 10, 2003, 9:00) ^ time < DATE(May, 10, 2003, 11:00)
8     Operation OpenExam{
9       Precondition #OpenExam.start=0
10      Action { ans=new OBJECT AnswerBook;Grant exam readPaper }
11     Operation Write {
12       Precondition #OpenExam.finish>0
13       Action Grant ans writeAnswer }
14     Operation Submit {
15       Precondition #Write.finish>0
16       Action ChangeOwner(ans, Checker) }}
17   Role Checker {
18     AdmissionConstraints #members(thisRole)<1 ^ member(thisUser, parentActivity.Grader)
19     Operation Grade {
20       Precondition #Candidate.Submit.finish=1
21       Action Grant ans setGrade }}
22 }

```

Figure 10: Specification of *ExamSession* activity template

During instantiating a child activity template, members to be assigned to the roles in the child activity can be specified as actual parameters. The formal parameters for the activity template specifies the roles that must be assigned members during instantiation, as discussed in Section 3.7.1. In Figure 8 (lines 16-17), when an examinee invokes the *StartExam* operation, an instance of the *ExamSession* template is created, and the participant creating the instance is dynamically assigned to the *Candidate* role. If member assignment is not specified in the actual parameters, the invoker of the operation can assign members to the required roles complying with the role's admission constraints, if any.

### 3.7.5 Role Activation Constraints

These are used to specify common preconditions for all operations defined for the role. In Figure 10 (line 7), an activation constraint, where the candidate can perform an operation only during the designated time for the exam, is specified.

```
time>DATE(May, 10, 2003, 9:00) ^time<DATE(May, 10, 2003, 11:00)
```

A cardinality constraint, which specifies a minimum number of members that must be present before any role operation can be performed, is specified as an activation constraint. 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.

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

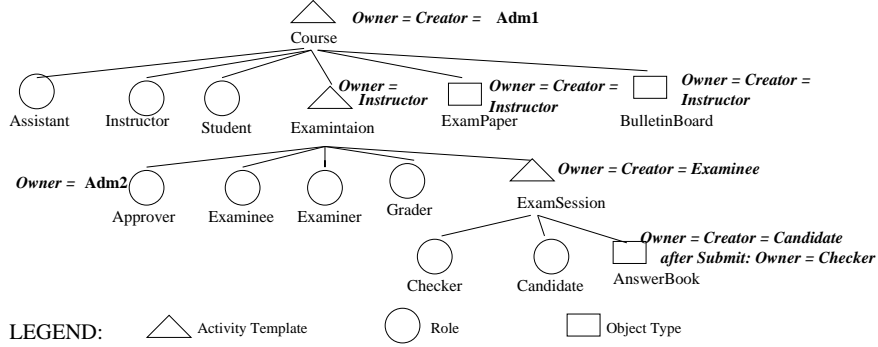


Figure 11: Owner-assignments in the nested *Course* activity template specification

### 3.8 Meta Policy Specification

In this model, the collaboration designer can specify an *Owner* for every entity – activity instance, role, and object. Another meta role related to each activity and object is *Creator*, which contains as its member the user who instantiates the activity or creates the object. 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. *Creator* can be specified as an owner. Only a role defined in the ancestor activities can be specified as an owner for an activity or a role. This ensures that no circular ownership relation exist among owners. For an object, a role defined in the ancestor activities as well as the same activity where the object is created 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.

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

Figure 10 presents the *ExamSession* activity template with owner assignments. In Figure 10 (line 1), *Creator* is specified as the owner of an *ExamSession* activity instance, and only the member of the *Creator* role can join the *Candidate* role (line 5). Within an exam-session, the candidate can create an *AnswerBook* object (line 8) and becomes the owner of the object, by default rules. After the candidate has taken the exam, he should no longer be trusted to manage the answer-book. In Figure 10 (lines 14-16), after the *Submit* operation, the ownership is transferred to the *Checker* role.

Figure 11 illustrates owner assignments of the entities in the nested specification of the *Course* activity template as presented in Figure 1. Figure 12 presents the resulting ownership relations among the entities in Figure 11. These ownership relations are derived based on the above membership rules of the *Owner* roles.



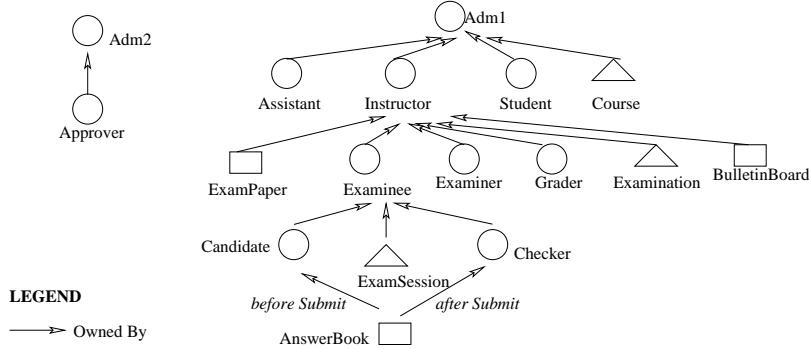


Figure 12: Owner hierarchies derived from Figure 11

In Figure 11, two roles  $Adm1$  and  $Adm2$  are specified as parameters.  $Adm1$  represents the *Creator* of the top level *Course* activity, and  $Adm2$  represents the *Owner* of the *Approver* role. In a cross-domain collaboration, participants of the domain that initiates an activity may not be trusted to manage some roles in the activity. For example, in an auditing activity, members of the auditor role must be managed by the auditing firm and cannot be managed by the audited firm. In our example *Course* activity, a similar requirement is specified, which requires that the *Approver* role's membership must be managed by a role in an outside organization. The role in the outside organization is specified as the parameter  $Adm2$ .

In Figure 11, by default rules, as the  $Adm1$  role is the creator, it is the 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 rules, *Instructor* role is the owner of the *Examiner*, *Grader*, and *Examinee* roles. Moreover, as *Creator* is assigned as the owner for the *ExamSession* template, the examinee who initiates an exam-session is the owner of the session.

In Figure 12, the ownership relations form hierarchical structures. This is due to the restriction that only the roles declared in ancestor activities can be assigned as the owner of a role or an activity. There are two owner-hierarchies under  $Adm1$  and  $Adm2$  as members in these two roles are assigned by two different organizations. In the execution environment, an owner is associated with a *trusted server*, which signifies a trusted protection mechanism for enforcing polices. Details of the runtime execution environment can be found in [Ahm04].

## 4 Properties for Verification

In the previous section, we have presented a role based specification model for coordination and security requirements in decentralized CSCW systems. The model is illustrated with specifications of different coordination and security requirements. In this section, we present different aspects of coordination and security requirements that a collaboration designer may specify as properties for verification during designing a collaboration specification. Verification of a collaboration specification has two distinct motivations. First, it has to ensure that the specification is not inconsistent. Second, it has to ensure that security and coordination requirements are satisfied by the specification.

## 4.1 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 specification of two inter-dependent role operations represents a deadlock, where none of the operations can be performed.

**Operation Op1 Precondition**  $\#(\text{Op2.finish}) = 1$

**Operation Op2 Precondition**  $\#(\text{Op1.finish}) = 1$

Such incorrect specifications result from inconsistent requirements or wrong specification of requirements.

## 4.2 Task-Flow

Task-modeling, represented as dependency among operations and activities, is an integral part of a collaboration specification. In the specification model, task-flow requirements, i.e, permissible sequence of operations, are specified with roles. In CSCW systems, the collaboration 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* [CH74] 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 presented in Figure 2, 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* operation, and the number of exam-session activities has to be equal to the cardinality of the *Examinee* role before the *Examination* terminates.

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

## 4.3 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**  $\text{!member}(A)$

**Role C Admission Constraints**  $\text{member}(A) \wedge \text{member}(B)$

Though there are several role related requirements specified for the example in Figure 1, we choose the following two role constraints (RC) 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.*

## 4.4 Information Flow and Confidentiality

As RBAC is “policy neutral” [San98], we can model information flow constraints by classifying roles with disjoint members with implicit security 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 information flow (IF) requirements.

**IF1** *A member of the examinee role cannot access the content of the question paper before start of his/her own exam session.*

**IF2** *Before the submission of the grades, identity of a candidate should not be known to the member of the assistant role who grades the candidate’s answer book.*

## 4.5 Integrity and Access Leakage

In the role-based collaboration model, access rights can only be leaked if unauthorized users can join a role. Integrity policies can be expressed as desired properties to check that access rights are not leaked. In our example, the collaboration designer specifies the following integrity requirement as an access leakage (AL) property.

**AL1** *A participant of the examinee role can modify his/her answer book only before the end of his/her exam-session.*

# 5 Verification Methodology

Here, we present a verification methodology using model checking to ensure that a specification is correct and satisfies various coordination and security requirements.

## 5.1 Challenges in Model Checking

SPIN [Hol03] is a model checker based on an automata theoretic approach. 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** ( $\square$ ), **eventually** ( $\diamond$ ), and **until** ( $\mathcal{U}$ ). There are several challenges in model checking:

### 5.1.1 State Space Explosion

The well-known problem of model checking is the state space explosion, i.e. exponential growth of state space. The search space for the PROMELA model for a small collaboration can be very large. To overcome this problem, 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 excluded from the verification model and 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. 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.

For efficient verification, we have developed four verification models based on the aspects of global requirements to be checked. The *Task Model* is developed to verify reachability of operations and task-flow constraints. The *Role Model* is for role constraints. The *Information Flow Model* is to verify properties related to information flow. Lastly, the *Owner Model* is concerned with the properties that can be affected by administrative privileges. Each model only includes components representing the aspects of a specification that the model verifies. The models are developed incrementally by adding and removing components that maintain state needed to verify a specific property.

### 5.1.2 Model Extraction

In our approach, the collaboration specification in XML is currently manually converted to the verification language of the model checker, i.e., PROMELA. However, additional components are added to the model to verify properties related to information flow, access leakage, and owner assignments. Similarly, the given requirements are converted to LTL expressions that refer to variables in the verification model.

To reduce state space, internal data structure also requires abstraction. For example, in the *Course* activity, if user  $C$  is an initial assignment to the *Assistant* role,  $C$  can eventually join the *Grader* role. It can be expressed as a correctness requirement for the *Grader* role as the following expression using LTL.

$\square \diamond \text{member}(C, \text{Grader})$

In our implementation, the verification model maintains a bit vector for users, where a bit the signifies presence of a user in a role. During initialization, roles and users require tracking are marked. 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  $C$ ’s presence in the *Grader* role.

$\square \diamond \text{member\_present}[j]$

To express various properties in LTL, several primitive predicates are supported, and additional predicates can be supported, if required for a desired property. Examples of the predicates include,

- *member* (*user*, *role*): the *user* is a member of the *role*.
- *event*(*event-type*, *user*): the *user* has triggered the specified type of event.
- *member* (*user*, *role*, *activity*): the *user* is a member of the *role* within the *activity*.
- *event*(*event-type*, *user*, *activity*): the *user* has triggered the specified type of event within the *activity*.
- *count*(*event*, *n*): the count of *event* is equal to *n*.

### 5.1.3 Inter-Dependent Requirements

Another problem is the inter-dependency of the verification requirements. Modification of the specification to comply with a requirement may violate any of the previously verified requirements. This results in a large number of iterations and complexity in managing the requirements [KS98]. During designing a collaboration specification, the verification methodology finds either an inconsistent specification, where a specified operation can never be executed or a role can never have a member, or an incorrect specification, where specified constraints do not satisfy a given requirement. In the first case, the requirements have to be modified to have a consistent specification. For the second case of incorrect specification, either the specified constraints or the requirements have to be modified. To reduce the iteration in the verification process, we have developed a verification methodology that follows a precedence among the properties it checks. The idea is to ensure the correctness and consistency of role and operation related requirements, before verification of information flow and administrative privilege related properties.

In our verification methodology, in the first step, reachability of operations and operation precedence related constraints are verified with the *Task Model*. Then, the role constraints are verified with the *Role Model*. These two models verify independent aspects of a specification and can be executed in either order. Next, verification with the *Information Flow Model* without administrative privileges checks if confidentiality properties are violated. If a verification fails, some constraints in the specification may require modification. In case a role constraint is modified, the verification steps with the *Role Model* and the *Information Flow Model* are repeated. For modification of a coordination constraint, the verification steps with the *Task Model* and the *Information Flow Model* are repeated. At the last step, the *Owner Model* verifies requirements, such as role constraints, information flow, or access leakage that can be violated due to incorrect assignment of administrative privileges. If a verification fails and only owner assignments are modified, the requirements are re-verified with the *Owner Model*. Modification of other constraints requires re-verification with the models that are related to the constraints.

### 5.1.4 Unbounded Participants

In verification models, to check the user-related properties, the presence of users in roles is modeled. A collaboration specification can be verified with a user-defined bound on the number of participants. As it can result in state-space explosion, such a bound cannot be very large. Hence, we need a bound on the number of participants for verification, such that it does not miss a possible execution trial, which may violate a property. We have developed a procedure to find a bound on the number of users for verification of a specification give a set of requirements. The bound on the number of users is determined for a subset of the roles that do not have any prerequisite membership constraints. That is, admission to these roles do not depend on the users' memberships to other roles. These roles without any prerequisite membership constraints are termed *initial assignment roles*. Only the users assigned to these roles can join or be admitted to other roles in a collaboration specification. In the example *Course* specification, *Student*, *Assistant*, *Instructor*, and *Approver* are initial assignment roles. The bound on the number of users for a *Course* activity is 5 with 2 in *Student* role, 1 in the *Assistant*, and 2 in the *Instructor* and *Approver* role. The details of the procedure that finds this number can be found in [Ahm04]. For the verification process presented in this paper, we assigned user identity *A* and *B* to *Student*, *C* to *Assistant*, and *D* and *E* to *Approver* and *Instructor* roles.

In the following subsections, each of the models are discussed in details including the aspects of a specification that are abstracted in the model and the expressions of the corresponding properties

```

1 proctype ExamSession_Activity( ) {
2   bit Write_finish=0, OpenExam_start=0, OpenExam_finish=0, Submit_finish=0, Grade_finish=0;
3   do
4     ::Grade_finish == 0 ->
5     if
6       /* OpenExam */
7       :: atomic { OpenExam_start == 0 -> OpenExam_start = 1; } OpenExam_finish = 1;
8       /* Write */
9       :: 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 }

```

Figure 13: Task Model in PROMELA for *ExamSession* activity in Figure 8

in LTL for verification. The models are presented in the above mentioned order of the verification methodology to illustrate a case study using the *Course* collaboration specification and the requirements in Section 4.

## 5.2 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.

Figure 13 shows the *Task Model* in PROMELA of the *ExamSession* activity specification, as presented in Figure 8. This model only includes the components that are required to verify operation precedence related properties. In this verification model, each activity is a process (line 1) and multiple instances of the process can be created. Within a process each operation's precondition is modeled as a guarded statement (lines 7, 9, 11, and 13). When the guard becomes true, the statement that follows after the arrow ( $->$ ) is executed in a non-deterministic step. The *atomic* statement (line 7) 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 condition is satisfied the global variable *ExamSession\_finish* is incremented.

An exhaustive verification run on this model reports unreachable code, pointing out the operations, which are unreachable. In addition, the path expressions for the task-flow requirements are converted to LTL expressions. The path expressions are converted to LTL expressions. In the following, only the response properties of the *Examination* activity, as discussed in Section 4.2, are presented in LTL.

```

□( Examination.start → ◇ Examiner.SetPaper.start)
□( Examiner.SetPaper.finish → ◇ Approver.ApprovePaper.start)
□( Approver.ApprovePaper.finish → ◇ Student.ExamSession.start)
□( count(Student.ExamSession.finish, #member(Examinee)) →
   ◇ Examination.finish)

```

In the verification run, if any of these properties related to operation precedence is not satisfied, a trace of the counter-example is provided by the model checker.

### 5.3 Verification Model for Role Constraints

Goal of the *Role Model* is to ensure that (1) all roles can have members, and (2) role admission and validation constraints can be satisfied. This verification model includes only components related to the role membership management aspects, such as static and dynamic role member assignment, role admission, and validation constraints. As this model is not concerned with policies expressed as part of operation preconditions, role operations are excluded from this verification model.

Based on the initial members assigned, the exhaustive search by the model checker reports unreachable code, in turn, pointing to the role that cannot have a member. For a large collaboration, for faster verification, a specific counter-example is searched by expressing the property, e.g, a role will eventually have a member.

To facilitate the designer to express various types of role constraints, conversion functions for role constraints to LTL expressions are provided. For example, the *static separation of duties* that a user  $x$  cannot be a member of two roles  $r1$  and  $r2$  is expressed with the following LTL expression using the primitive predicates. In the verification run,  $x$  is replaced by user identities, and  $r1$  and  $r2$  are replaced with role names.

$$\text{SSOD}(r1, r2) := !\diamond ( \text{member}(x, r1) \ \&\& \ \text{member}(x, r2) )$$

*Case Study – Verification of RC1:* *RC1* is a *static separation of duties* requirement, i.e., a member of a *Checker* role cannot be a member of *Candidate* role. An optimization of this process is to verify the property based on the only possible member in the *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.

$$\text{SSOD}(\text{Checker}, \text{Candidate}) := !\diamond ( \text{member}(C, \text{Checker}) \ \&\& \ \text{member}(C, \text{Candidate}) )$$

*Case Study – Verification of RC2:* Knowing that users  $A$  and  $B$  are initial members of the *Student* role, the requirement *RC2* is expressed as below.

$$!\diamond ( \text{member}(A, \text{Candidate}, \text{es1}) \ \&\& \ !\text{event}(\text{ExamSession\_start}, A, \text{es1}) )$$

The requirement is specified by negating the fact that eventually user  $A$  is a member of the *Candidate* role without starting the *ExamSession* instance  $\text{es1}$ . In this expression an activity  $\text{es1}$  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 for this verification. This requirement was satisfied.

### 5.4 Verification Model for Information Flow

Several confidentiality properties, such as noninterference, noninference, and non-deducible, have been formalized [ZL97]. However, in our verification model only explicit information flow 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.,  $o1 \rightarrow o2$ .
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.,  $s1 \rightarrow s2$ .

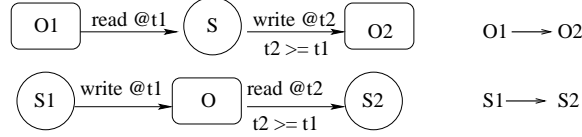


Figure 14: Information flow: object to object, subject to subject

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 as parameters to method invocations or object creations. One can also rely on explicit declaration of methods in these two categories, *read* and *write*, by object designers. To express properties related to information flow, the verification model supports additional predicates. The predicate  $knows(subject, object)$  signifies that the *object* content has passed to the *subject*. Similarly, the predicate  $knows(subject, members(role, activity))$  signifies that the *subject* knows the identities of the members of the *role* in the *activity*.

This verification model is extended from the *Task Model*. As oppose to the *Role Model*, which includes components representing role membership related operation such as *join* and *admit*, the information flow model abstracts only possible membership in each role using global data structures. *Case Study – Verification of IF1*: Knowing that users *A* and *B* are initial members of the *Student* role, we express the information flow requirement *IF1*, in Section 4.4, as “user *A* of the examinee role cannot access the content of the exam paper before start of his own exam session”. This requirement is expressed as below,

$! \diamond (knows(A, ExamPaper) \ \&\& \ !event(ExamSession\_start, A))$

It is specified by negating the fact that eventually user *A* knows the content of the *ExamPaper* without starting his *ExamSession*. Steps through which the original specification was modified to comply with this requirement are discussed below.



Figure 15: Trace of a counter-example: *Examiner* leaked *ExamPaper*

- In our initial run, with users *A* and *B* to *Student*, *C* to *Assistant*, and *D* and *E* to *Instructor* and *Approver*, a counter-example was found, as presented in Figure 15. In the trace, *D* being a member of the *Examiner* had access to the *ExamPaper*. However, *D* also being a member of the *Instructor* within the *Course* activity wrote the *ExamPaper* to the *BulletinBoard*. User *A*, a member of the *Examinee* and the *Student* roles, accessed this content through the *BulletinBoard* before starting his exam-session. That is the *Instructor* leaked the *ExamPaper* to the examinees before the start of their exam-sessions. To encode that such an act would not be performed by the *Instructor*, we provided this fact to the model as a tuple  $!write(Instructor, ExamPaper, BulletinBoard)$ , which meant that *Instructor* would not *write ExamPaper* content to the *BulletinBoard*.

- In the second run, as shown in Figure 16, candidate *B* initiated his own *B’s ExamSession* and wrote the content of the *ExamPaper* to the *AnswerBook*. Checker *C*, who had no direct access to the *ExamPaper*, accessed it from *B’s AnswerBook*. Checker *C* leaked this content through the *BulletinBoard* to examinee *A*, who had not initiated his exam-session. A fact that *Checker* would not transfer *AnswerBook* content to the *BulletinBoard* was provided to the model.



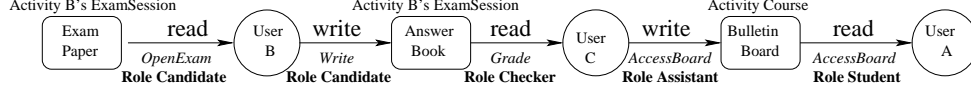


Figure 16: Trace of a counter-example: *Checker* leaked *ExamPaper*



Figure 17: Trace of a counter-example: *Candidate* leaked *ExamPaper*

- The next verification run, as shown in Figure 17, found another counter-example where candidate *B* was able to leak the content of the *ExamPaper* through the *BulletinBoard* before user *A* had started his own *ExamSession*. To preserve this property of information flow, the *Student* role's privileges on the *BulletinBoard* were revoked during the *Examination* activity. This was accomplished by adding a coordination constraint on the operations of the *Student* role accessing the board.

*Case Study – Verification of IF2:* The confidentiality requirement *IF2*, with user *C* being a member of the *Assistant* role, is expressed as below.

```
!◇( !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, and *Candidate* role member's identity, i.e., *A*'s identity is known to *C* before the *Grade* operation is finished by *C*. A counter example was found where the candidate leaked his identity through the *AnswerBook* object, and the checker was able to access the identity during grading. The fact that *Candidate* would not perform such an action was provided to the model. Hence, *IF2* was satisfied.

## 5.5 Verification with Owner Assignment

Verification of security requirements with *Owner*'s administrative privileges has two concerns. First, participants in *Owner* role are *trusted*; however, security requirements can be violated due to their extended privileges for being in the *Owner* roles. Second, participants in *Owner* role may not be trusted and the verification process has to ensure that sensitive security requirements are not violated by untrusted owners. Hence, there are two *Owner Models* for *trusted* and *untrusted* owners.

### 5.5.1 Trusted Owner Model

The extended privileges for an *Owner* role are: (1) the owner of a role can view identities of the role members, and (2) the owner of an object can read/modify it without any restriction. These extended privileges can result in violation of information flow and access leakage properties. Consequently, there are two *Trusted Owner Models*: (1) to verify information flow requirements the *Information Flow Model* is extended with owners extended privileges on role membership information; (2) to verify access leakage properties, the components for information flow are replaced with components representing object access including unrestricted access by owners. In this model, to express access leakage properties an additional predicate is added. This predicate *access(permission, object\_type, user)* signifies that the *user* has the specified *permission* on an instance of the *object\_type*.

*Case Study – Verification of IF1, IF2:* In the next step of our verification process, the information flow properties IF1, IF2 were satisfied.

*Case Study – Verification of AL1:* The requirement AL1 is related to access leakage that the *write* privilege to the *AnswerBook* must be revoked when *ExamSession* terminates, which is expressed as:

```
!◇( event(ExamSession_finish, A) && access(write, Answer_Book, A))
```

The requirement is specified by negating the fact that eventually there is a state where *A*'s *ExamSession* activity has been terminated and *A* has *write* access to an *Answer\_Book*. This requirement was satisfied.

### 5.5.2 Untrusted Owner Model

In our specification model, an owner is trusted to enforce entity-specific policies, i.e., a role owner enforces role membership management policies, an object owner enforces object access policies, and activity owner enforces activity management policies. In the *Untrusted Owner Model*, for modeling of untrusted owners, we rely on types of policies that can be violated by the owners. In the untrusted owner model, entities owned by the untrusted owners can be *misbehaving*. To derive verification models for misbehaving entities, the violation of 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. These behaviors are implemented by removing the role constraints for a role that results in admission of all possible participants in a role and generation of all possible invalid query results.

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. If a misbehaving entity is the notifier of coordination events, the entity is modeled as generating false coordination events or omitting coordination events. These behaviors are implemented by removing operation preconditions, and thus resulting in non-deterministic generation of operation events. For each subscriber of the operation event, an individual event variable is maintained. These variables are updated in non-deterministic steps that result in targeted omission of events.

Consequently, there are two *Untrusted Owner Models* extended from the two *Trusted Owner Models*, each incorporating the above two behaviors.

A 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 roles that may not be trusted for a requirement. Once the initial 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, 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 requirement with *Untrusted Owner Model*.

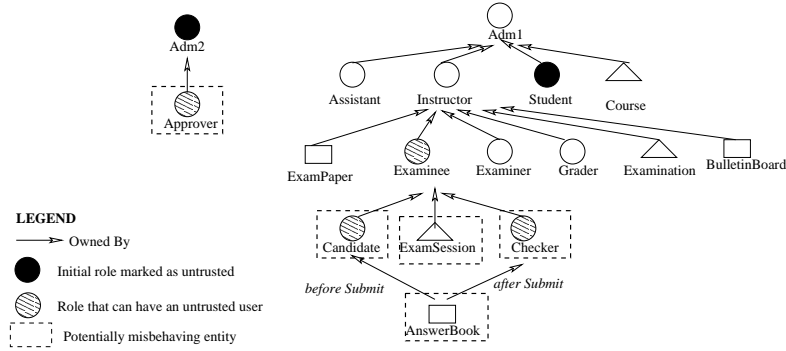


Figure 18: Potentially misbehaving entities derived based on a trust assignment in Figure 12

- Step 1: Identify the potentially misbehaving entities to be verified.
- Step 2: Among the potentially misbehaving entities, an entity in the scope of the inner most activity template is selected and modeled as misbehaving. As mentioned earlier, an entity misbehaves by either (1) violating role constraints or (2) violating operation preconditions.
- Step 3: 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 re-verified with the corresponding verification models.
- Step 4: 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. Repeat steps 2, 3, and 4 until all the potentially misbehaving entities are selected or all the requirements are verified.

Due to not enforcing operation precondition, false events can be generated enabling the precondition of another operation. This in turn can violate a coordination requirement or dynamic access control policy. This types of violation can be prevented by adding the precondition of the misbehaving operation as a precondition for the effected operation. A second type of violation of requirements can result from not enforcing preconditions is the omission of events to targeted subscribers, which may cause inconsistency in coordination policies. This type of violation 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.

*Case Study – Verification with Untrusted Owner:* For our example case study, the collaboration designer assigned members of the *Adm2* and *Student* roles as untrusted to enforce any policies, as presented in Figure 18. As members of the student role can join the *Examinee* and *Candidate* roles, untrusted users can become members of these roles. Based on unrestricted admission of the untrusted members of the *Owner* roles, potentially misbehaving entities were *Approver*, *ExamSession*, *Candidate*, *Checker*, and *AnswerBook*. Among the entities, the *Checker*, *Candidate*, and *Answer-Book* are defined in the inner most *ExamSession* activity. We chose the *Checker* role as our first potentially misbehaving entity and found that the sensitive requirement *RC1* was violated as the role constraints for the *Checker* role were not enforced. Next, the *Candidate* role was selected and the sensitive requirement *RC2* was violated as the *Candidate* role being misbehaving admitted any

user to the role. Next, the *AnswerBook* was selected, and the sensitive requirement, *AL1* failed as the *Candidate's* access to the *AnswerBook* was not revoked by the misbehaving *AnswerBook* after the end of the exam-session.

Next, we assigned a trusted role *Grader* instead of *Examinee* as the owner of the *ExamSession*. Based on the owner rules, the *Grader* becomes the owner of the nested *Checker* and *Candidate* roles. As owner assignments had changed, all the security requirements are re-verified with the owner models. With the *Grader* being the owner, the requirement *IF2* that the *Checker* role must not know participants' identities of the *Candidate* role was violated as *C* in the *Checker*, being a member of the owner *Grader*, had access to the *Candidate* role's membership information. Finally, we assigned the *Examiner* as the owner of the *Candidate* role.

## 5.6 Discussion

Our goal of the verification process is to ensure that the specification can satisfy the given requirements [GGJZ00]. We have presented several verification models that implement a collaboration specification by incrementally adding components related to all the properties of the specification. The primary difference among these models is the additional components that are added to maintain the verification property specific information. Moreover, specific facts about trusting members of a role for not performing actions that may result in policy violation is added in the models. As such listing can be exhaustive, only the facts that are required during verification process are specified.

To reduce search-space, property specific models are developed. A false negative due to incomplete model abstraction provides guidance toward developing the model. The resultant counter-example provides clues on the model abstraction errors that cannot exist in a concrete system. On the other hand, a false positive due to unsound model abstraction is a known challenge [Hol03]. This fails to find a counter-example that may exist in a concrete system. The only guarantee that can be provided for a verified specification with a property is that there is a concrete implementation that can satisfy the corresponding requirement.

## 6 Related Work

The work presented in this paper relates to role-based security models and verification of security policies. Similar to *activity* in our model, other role-based models have defined different concepts for the context of a role, such as *domain* in [LS97], *role template* in [GI97], and *authorization template* in [HA99]. Only a few of these models can express history-dependent security policies. None of these addresses user-assignment in administrative roles. We have developed a role-based specification model for security and coordination policies, including administrative policies, for distributed CSCW systems. In comparison with other research in role-based models for decentralized management [SBM99, OS02, BMY02, CL03], we have developed an integrated administrative model, which assigns different classes of administrative privileges to roles within a collaboration.

In RBAC, safety of various role-based constraints, such as separation-of-duties, have been analyzed with logical expression [BF99, AS00, HA99, JSS97] and graphical models [JT01, Osb02, KMPP02]. Only few of these models can specify and verify correctness of constraints that are based on task. The verification in most of these existing research is performed in centralized management of systems. We have utilized finite state model checking for verification of security requirements in decentralized management of CSCW systems. Model checkers, which support LTL property,

are used for verification of other system properties, such as workflow processes [EW02], security protocols [MS02], and program analysis [CDH<sup>+</sup>00].

Others, e.g., [FG97, Sch96, VS00], have utilized formal methods for verification of security properties, mainly various types of confidentiality properties. 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” [ØP97] or labeling data [ML00]; however, the execution environment is assumed to be entirely trusted. In contrast, our collaboration specification can be verified by assigning facts regarding trust on specific roles. For untrusted owners, the policy enforcement mechanisms in distributed hosts is modeled as trying to violate specific policies by omitted or falsifying coordination event. Similar requirements on securing event causality are addressed in [RG95].

Our goal is to design collaboration specification with safe assignment of administrative privileges. Similar goals are expressed by other researchers. E.g., role modeling using software engineering methodology is presented in [ES01]. The verification methodology presented in this paper supports a collaboration designer to specify security constraints ensuring that a specification is correct and consistent given a set of requirements.

## 7 Conclusions

The work presented in this paper is motivated by rapid realization of collaboration systems based on pre-generated specifications. The primary contribution of this paper is a methodology for specification and verification of security requirements in decentralized CSCW systems. We have developed a role based specification model to express different security and coordination requirements, including administrative security requirements, in distributed collaboration systems. Trust classes are defined to assign administrative privileges to participants taking into account the the presence of untrusted users due to different and independent security domains. Finite-state model checking techniques are utilized for static verification of a collaboration specification. Based on the aspects of the requirements, we have presented incremental verification with four verification models to handle problems related to state space explosion and inter dependency of the requirements.

## References

- [Ahm04] Tanvir Ahmed. *Policy-Based Design of Secure Distributed Collaboration Systems*. PhD thesis, University of Minnesota, November 2004. Available at <http://www.cs.umn.edu/Ajanta/publications.html>.
- [ARH98] Alfarez Abdul-Rahman and Stephen Hailes. A distributed trust model. In *Proceedings of the Workshop on New Security Paradigms Workshop*, pages 48 – 60, New York, 1998. ACM.
- [AS00] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207 – 226, November 2000.
- [BF99] Elisa Bertino and Elena Ferrari. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transactions on Information and System Security*, 2(1):65 – 104, February 1999.

- [BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Symposium on Security and Privacy*, pages 164–173, Los Alamitos, CA, 1996. IEEE Computer Society Press.
- [BMY02] Jean Bacon, Ken Moody, and Walt Yao. 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, November 2002.
- [CDH<sup>+</sup>00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of International Conference on Software Engineering*, pages 439 – 448, 2000.
- [CH74] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In *Operating Systems, International Symposium, Rocquencourt*. Lecture Notes in Computer Science vol.16, Springer Verlag, April 1974.
- [CL03] Jason Crampton and George Loizou. Administrative Scope: A Foundation for Role-Based Administrative Models. *ACM Transactions on Information and System Security*, 6(2):201 – 231, May 2003.
- [CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, Los Alamitos, CA, 1987. IEEE Computer Society Press.
- [Den93] Dorothy E. Denning. A new paradigm for trusted systems. In *Proceedings on the 1992-1993 Workshop on New Security Paradigms*, pages 36 – 41, New York, August 1993. ACM.
- [DTT93] S.A. Demurjian, T.C. Ting, and B. Thuraisingham. User-role based security for collaborative computing environments. *Multimedia Review*, 4(2):40–47, Summer 1993.
- [ES01] Pete Epstein and Ravi Sandhu. Engineering of Role/Permission Assignments. In *17th Annual Computer Security Applications Conference*, December 2001.
- [EW02] Rik Eshuis and Roel Wieringa. Verification Support for Workflow Design with UML Activity Graphs. In *Proceedings of International Conference on Software Engineering*, pages 166 – 176, New York, 2002. ACM.
- [FCK95] David Ferraiolo, Janet Cugini, and Rick Kuhn. Role Based Access Control (RBAC): Features and Motivations. In *IEEE Annual Computer Security Applications Conference*, pages 241 – 248, Los Alamitos, CA, 1995. IEEE Computer Society Press.
- [FG97] Riccardo Focardi and Roberto Gorrieri. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.
- [GGF98] V.D. Gligor, S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings IEEE Symposium on Security and Privacy*, pages 172–183, Los Alamitos, CA, 1998. IEEE Computer Society Press.

- [GGJZ00] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May/June 2000.
- [GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 153 – 159, November 1997.
- [GS87] Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Information Systems*, 5(2):187–211, 1987.
- [HA99] Wei-Kuang Huang and Vijayalakshmi Atluri. SecureFlow: A Secure Web-enabled Workflow Management System. In *ACM Workshop on Role-based Access Control*, pages 83 – 94, New York, 1999. ACM.
- [Hol03] Gerard J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional, 2003.
- [JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31 –42, 1997.
- [JT01] Trent Jaeger and Jonathon E. Tidswell. Practical Safety in Flexible Access Control Models. *ACM Transactions on Information and System Security*, 4(2):158 – 190, May 2001.
- [KMPP02] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security*, 5(3):332 – 365, August 2002.
- [KS98] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. John-Wiley & Sons, Chichester, New York, 1998.
- [LS97] Emil C. Lupu and Morris Sloman. Reconciling Role-Based Management and Role-Based Access Control. In *ACM Workshop on Role-based Access Control*, pages 135–141, New York, 1997. ACM.
- [McL94] J. McLean. Security Models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.
- [ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
- [MS02] P. Maggi and R. Sisto. Using SPIN to Verify Security Protocols. In *Proceedings of 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318*, pages 187–204, 2002.
- [ØP97] Peter Ørbæk and Jens Palsberg. Trust in the  $\lambda$ -calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.
- [OS02] Sejong Oh and Ravi Sandhu. A Model for Role Administration Using Organization Structure. In *ACM Symposium on Access Control Models and Technologies*, pages 155 –162, New York, 2002. ACM.

- [Os02] Sylvia L. Osborn. Information Flow Analysis of an RBAC System. In *ACM Symposium on Access Control Models and Technologies*, pages 163 – 168, New York, 2002. ACM.
- [RG95] M. Reiter and L. Gong. Securing Causal Relationships in Distributed Systems. *The Computer Journal*, 38(8):633–642, 1995.
- [RV77] P. Roberts and J-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proceedings of IFIP Congress*, pages 981–986, Amsterdam, 1977. North-Holland.
- [San98] Ravi Sandhu. Role activation hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*, pages 33 – 40, 1998.
- [SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105 – 135, February 1999.
- [Sch96] Steve Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, pages 174 –187, Los Alamitos, CA, 1996. IEEE Computer Society Press.
- [SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control*, pages 47–63, New York, July 2000. ACM.
- [SZ97] R.T. Simon and M.E. Zurko. Separation of duty in role-based environments. In *10th Computer Security Foundations Workshop*, pages 183 –194, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [VS00] Dennis M. Volpano and Geoffrey Smith. Verifying Secrets and Relative Secrecy. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–276, New York, 2000. ACM.
- [YKB93] R. Yahalom, B. Klein, and T. Beth. Trust relationships in secure systems-a distributed authentication perspective. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 150 –164, Los Alamitos, CA, 1993. IEEE Computer Society Press.
- [ZL97] A. Zakinthinos and E.S. Lee. A General Theory of Security Properties. In *IEEE Symposium on Security and Privacy*, pages 94 –102, Los Alamitos, CA, 1997. IEEE Computer Society Press.