Specification and Verification of Security Requirements in a Programming Model for Decentralized CSCW Systems

TANVIR AHMED
and
ANAND R. TRIPATHI
University of Minnesota, Minneapolis

We present in this paper a role-based model for programming distributed CSCW systems. This model supports specification of dynamic security and coordination requirements in such systems. We also present here a model checking methodology for verifying the security properties of a design expressed in this model. The verification methodology presented here is used to ensure correctness and consistency of a design specification. It is also used to ensure that sensitive security requirements cannot be violated when policy enforcement functions are distributed among the participants. Several aspect-specific verification models are developed to check security properties, such as task-flow constraints, information flow, confidentiality, and assignment of administrative privileges.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms: Management, Design, Security, Verification

Additional Key Words and Phrases: Security policy specification, Role based access control, Methodology for access control policy design, Finite-state based model checking

1. INTRODUCTION

CSCW (Computer Supported Cooperative Work) systems are designed to support cooperative activities involving a group of users performing tasks related to some shared objectives. Examples of such systems include online conferencing, collaborative design and development, and workflow environments. Management of distributed CSCW systems for such applications often needs to be decentralized, when such systems are designed for ad hoc integration of users from different organizations or peer groups. The focus of our work is on building secure decentralized CSCW systems from their high level specifications.

This work was supported by National Science Foundation grant 0082215 and 0411961.
This article extends [Tripathi et al. 2003] and [Ahmed and Tripathi 2003].
Author's address: Tanvir Ahmed and Anand R. Tripathi, {tahmed,tripathi}@cs.umn.edu, Department of Computer Science, University of Minnesota, Minneapolis MN 55455

Security and coordination requirements in CSCW systems tend to be dynamic and context-based, depending on the execution state of the collaborative tasks and history of participants' actions. The coordination requirements are often weaved with access control concerns. Such requirements have been addressed in workflow systems to synchronize authorization and access control mechanisms with task-flow events [Sandhu 1988; Atluri and Huang 1996]. Moreover, role based access control (RBAC) models [Sandhu et al. 1996] have been found to be naturally useful in CSCW systems because of their intrinsic ability to model organizational structures [Greif and Sarin 1987; Demurjian et al. 1993]. Specification and enforcement of dynamic security and coordination requirements in role-based models is an important problem [Bertino et al. 1999; Huang and Atluri 1999; Ahn and Sandhu 2000].

Another challenge in specifying security policies for distributed CSCW systems is the expression of administrative level security requirements. A distributed CSCW system may require decentralized management as no single organization, site, or participant may be trusted to act as a "reference monitor" for the management and enforcement of all of the policies of the system. With decentralized management, the *ownership* and associated policy enforcement privileges for the various entities – roles and objects – in the shared workspace may be under the control of different participants. However, some participants may not correctly enforce the part of the policies that they are entrusted with, thus possibly resulting in violation of overall security requirements for the system. In our work, such participants are designated by the CSCW system designer as *untrusted* for some of the policy enforcement functions.

An important goal of our verification methodology is to ensure that in decentralized management of a CSCW system the assignment of ownership privileges for an entity to an untrusted participant does not result in violation of any sensitive security requirements. The goal of our verification methodology is to determine *safe* assignments of ownership privileges in a design to satisfy the given set of security requirements.

The primary contributions of this paper are twofold:

(1) Development of a role-based model together with a programming framework for specification of coordination and security requirements in distributed CSCW systems.
(2) Development of a verification methodology based on finite-state model checking using SPIN [Holzmann 2003] to ensure that a design expressed in this model satisfies a given set of requirements for coordination and security. The verification methodology is used to ensure the following kinds of properties in a design:
    —User interactions follow coordination and task-flow requirements;
    —Roles do not have conflicting or inconsistent constraints;
    —Confidential information cannot flow to unauthorized users;
    —No access rights can be leaked to unauthorized users;
    —Authorized information can be accessed;
    —Any dynamic constraints on accessing objects can be satisfied.

In the following section we discuss the contributions of our work in the context of other research in this field. Section 3 presents the dynamic security and coordination requirements in CSCW systems. The role-based model developed for secure distributed CSCW systems is described in Section 4. Based on this role-based model, a specification framework for distributed CSCW systems is presented in Section 5. Section 6 discusses the goals of the design verification process and presents the verification issues that arise due to decentralized policy enforcement in the presence of some untrusted roles. The issues in extracting the PROMELA model of a design and verification using SPIN are discussed in Section 7. Our verification methodology is presented in Section 8. Section 9 presents the conclusions of our work.

2. GOALS AND RELATED WORK

Our work has been driven by the goal of developing a programming framework for constructing secure distributed CSCW systems from their high level specifications [Tripathi et al. 2003]. We present here a role-based model that is used by the designer of a CSCW system for specifying its architectural design for integrating application level components and users, and specifying the policies for role-based user participation, coordination, and security. A middleware system automatically constructs the distributed runtime system for a given design. In the past, other researchers [Li and Muntz 1998; Corts and Mishra 1996] have also investigated this kind of approach for building distributed collaboration systems, but with the primary focus on coordination requirements. In contrast, our work addresses security requirements in CSCW systems, particularly with decentralized management.

In the specification model presented here, roles are defined in the context of an application rather than the global context of an organization. Others have also used similar concepts, defining the context of a role, such as *team* in [Thomas 1997], *domain* in [Lupu and Sloman 1997], and *role template* in [Giuri and Iglio 1997]. Similar to the RBAC model in [Sandhu et al. 2000], role permissions in our model represent object level operations. The RBAC model presented in [Bacon et al. 2002] for distributed autonomous domains, where roles resemble capabilities, has influenced our model for dynamic constraints on role membership and activation. An important aspect in which our work differs significantly from other role models is in regard to its support for specifying coordination constraints and shared privileges among role members. Such requirements for shared privileges in roles are discussed in [Lupu and Sloman 1997].

Integration of various different kinds of constraints in the RBAC model is discussed in [Sandhu et al. 1996]. Several researchers have developed models for specifying dynamic authorization constraints in RBAC [Bertino et al. 1999; Bertino et al. 2001; Ahn and Sandhu 2000; Huang and Atluri 1999; Jajodia et al. 1997], specifically motivated by higher level organizational policies such as separation-of-duties. A formal language for specifying authorization constraints in the RBAC96 model is presented in [Ahn and Sandhu 2000]. In [Bertino et al. 1999] static and dynamic constraints for separation-of-duties requirements in workflow systems are expressed as clauses in a logic programming language, which are enforced by a centralized mechanism. The focus of that work is primarily on specification, analysis, and enforcement of constraints. The language presented there is intended for system-level enforcement mechanisms, and not for application-level specification of constraints. In contrast, our work presents a specification framework which is intended for expressing complete architectural design of a CSCW system, with dynamic constraints specified as integral part of the design specification. TRBAC, a temporal RBAC model [Bertino et al. 2001], supports expression of temporal constraints for periodic enabling and disabling of roles. X-GTRBAC [Bhatti et al. 2005] is an XML-based policy specification language with a context-based RBAC model supporting dynamic fine-grained constraints for assigning users to roles, permissions to roles, and their activation. Our work also shares this goal of supporting dynamic context-based security requirements in role-based systems; however, instead of enterprise-wide policy specification our focus is on expressing these requirements in the design of a CSCW system.

For dynamic constraints, the notion of events and their integration with authorization mechanisms is a salient feature of our RBAC model. This is conceptually similar to *authorization template* [Atluri and Huang 1996] model where task-flow events are implicitly used for supporting dynamic authorization by the system using a Petri net model. In contrast, events are "first class" entities in our model and event-based predicates are specified explicitly. This event-based model can express different kinds of separation-of-duties requirements, and there is no need to include any policy-specific constructs as in [Crampton 2003]. Moreover, our model supports distributed and decentralized policy enforcement by the participants in the system. From the constraints specified in a design, a middleware generates the appropriate policy enforcement components, which may not be necessarily executed under a single user's control or managed in a centralized fashion [Tripathi et al. 2002].

Our model includes the concept of a meta-role, termed *owner*, associated with each object and role for policy enforcement related administrative privileges. It supports dynamic assignment of administrative rights based on the system state. In contrast to administrative RBAC models [Sandhu et al. 1999; Oh and Sandhu 2002; Crampton and Loizou 2003], where roles with administrative rights are separately defined, in our work, different participant roles of a CSCW system are entrusted with the *ownership* privileges for various entities in the system. The goal of our verification methodology is to ensure that the ownership assignments specified in a design are *safe* in the sense that an untrusted participant in any owner role would not be able to violate any sensitive security requirements.

For safety analysis and consistency checking of role-based constraints in workflow systems, a logic programming based approach is presented in [Bertino et al. 1999]. The focus of that work is on determining all valid execution paths for a workflow given a set of constraints, and enforcing constraints at runtime to ensure that only a valid path is taken. This problem is also addressed in [Crampton 2004] using a graph-based model. In contrast, our focus is on model-checking during the design phase, rather than at runtime, to verify safety properties as well as information flow properties of a system with decentralized control. Graph-based models have also been used to analyze safety of role constraints in RBAC models [Jaeger and Tidswell 2001; Nyanchama and Osborn 1999; Osborn 2002; Koch et al. 2002].

An approach based on type checking and data labeling in programming languages [Myers and Liskov 2000] has been developed for secure information flow in decentralized systems. However, trusted execution platforms are assumed to exists in the distributed execution environment. In contrast, we verify a design with the assumption that the members in certain roles are *untrusted* and they may violate policies in any arbitrary manner. During analysis, we model various aspects of such behavior of an untrusted participant in an owner role, such as bypassing operation preconditions or role admission constraints, and omitting or falsifying coordination events. Similar requirements on securing event causality are addressed in [Reiter and Gong 1995].

The RT framework [Li et al. 2002] is family of languages for trust management and policy specification for distributed authorization. This framework combines role-based access control and trust management model, with semantic foundations in logic programming using Constraint Datalog. A formal analysis for safety and availability based on several forms of access delegation models in this framework is presented in [Li et al. 2003]. Our research, on the other hand, is on a software engineering methodology for modeling, specifying, verifying, and realizing secure distributed collaboration systems. The CSCW systems expressed in our specification model have restricted structure in regard to the number of role types, the set of privileges associated with a role, and scope rules. This facilitates finite-state based model checking of security requirements including information flow. Moreover, the notion of trust in our work is mainly related to the designer's trust in various participant roles in regard to correctly enforcing policies under their control.

Our verification process is similar to research in finite state based model checking of workflow processes [Eshuis and Wieringa 2002; Janssen et al. 1998]. Using SPIN model checker, verification of workflow constraints is presented in [Janssen et al. 1998] and verification of RBAC constraints is presented in [Hansen and Oleshchuk 2005]. In contrast to these research, we utilize a model for collaboration environments for verification of coordination, role constraints, as well as security requirements, such as access leakage and information flow constraints. Similar to the approach used in finite-state based protocol verification [Maggi and Sisto 2002], we model *trusted* and *untrusted* participants in our verification procedures.

3. SECURITY REQUIREMENTS IN ROLE-BASED DISTRIBUTED CSCW SYSTEMS

We identify here dynamic security and coordination requirements in role-based decentralized CSCW systems. These requirements need to be expressed through appropriate constructs in a specification model.

3.1 Role Admission and Revocation Constraints

In distributed systems, the role admission related constraints need to support specification of conditions for granting or revoking role memberships [Bacon et al. 2002]. Role admission constraints specify the conditions that must to be satisfied for a user to be admitted in the role. These constraints can be based on: user's current or past membership in some "prerequisite" roles (for allowing admission) or in "conflicting" roles (for denying admission), history of past actions by a user, and role membership cardinality. The role admission constraints are also needed to enforce requirements related to static separation-of-duties. Because the role admission condition may not hold after a user has been admitted into the role, a role revocation condition is needed to verify the validity of a participant's current membership in a role.

3.2 Role-based Coordination Requirements

Coordination between participants in different roles within an activity is referred to as *inter-role* coordination, which is supported in several role based coordination systems [Li and Muntz 1998; Corts and Mishra 1996]. On the other hand, *intra-role coordination* is required when multiple members in a role need to coordinate among themselves. The role tasks can be considered as either shared or independent privileges for the role members [Lupu and Sloman 1997]. *Intra-role* coordination can be based on *independent*, *cooperative*, or *ad hoc* modes for role task execution by the members.

In independent participation, a role specific task responsibility is assumed individually by a member, irrespective of the presence of the other members in the role. For example, every member in the conference reviewer role has to independently write a review. On the other hand, when the members in a role are assuming task responsibilities cooperatively, they need to coordinate among themselves. Consider the example in [Lupu and Sloman 1997], where a hospital patient ward may have several nurses present in the role of *nurse-on-duty*. However, some medical procedure on a patient may be needed to be performed only once by any of the members. In some CSCW applications, the role members may interact in ad hoc and unstructured fashion, e.g. as in an unrestricted whiteboard sharing activity.

3.3 Dynamic Access Control Policies

Security requirements in CSCW systems tend to be dynamic in nature. Such requirements depend on the execution history of the collaborative tasks [Sandhu 1988; Atluri and Huang 1996]. They may also depend on temporal conditions – for example, certain tasks can only be performed during some specified time periods [Bertino et al. 2001] – or ambient conditions, such as the co-location of some users in some physical space [Sampemane et al. 2002]. The privileges assigned to a user in a role may change with time due to the actions of other participants. In some cases, permissions change due to the participant's own actions, such as making a final agreement on a document, after which the creator of the document may not have the right to modify it [Atluri and Huang 1996]. This includes situations where the ownership privileges for an object may change from one role to another. Another example is the requirement that a role operation be performed only when some minimum number of participants are members of that role. A broad range of of separation-of-duties requirements also tend to be dynamic [Sandhu 1988; Simon and Zurko 1997; Nyanchama and Osborn 1999] and they fall into the category of history-based access control policies.

3.4 Meta-level Security Policies

Secure management of a CSCW activity requires correct enforcement of the associated policies. For example, in managing a role, the admission and revocation constraints need to be enforced correctly. In a distributed CSCW system, there may not be a single participant, or a role, or a domain that could be trusted to serve as a *reference monitor* to enforce all of the security policies. Instead, it should be possible to designate for each entity (object, role, and activity) a role that can be trusted to correctly enforce its management functions. Specification of such meta-policies assigning administrative privileges to a role for managing specific entities in an activity needs to be supported. Such policies may also be dynamic, requiring change in the assignment of administrative privileges depending on the execution state of the activity.

3.5 Information Flow and Access Leakage

In a CSCW system, an important concern is to prevent information flow and access leakage to unauthorized users. Confidentiality requirements express such information-flow constraints. With decentralized policy enforcement and dynamic policies for admission of users into roles, it is important to ensure that any assignment of administrative privileges preserves integrity of access authorization and information confidentiality.

4. A ROLE BASED MODEL FOR CSCW SYSTEMS

We present here the central elements of the role based specification model for programming distributed CSCW systems. In our model, an activity is an abstraction of a collaboration or workflow task involving a set of users in various roles. These users perform collaborative tasks involving some shared objects/resources. In an activity, users are represented by their roles, and roles within an activity are assigned privileges to perform certain tasks. We term these role specific tasks as *operations*. An operation typically involves invocation of a method on an object defined within the activity or creation of a new activity.

4.1 Activity Template

An *activity template* defines a pattern for a CSCW activity. An activity is created and started by instantiating its template using a distributed middleware system [Tripathi et al. 2002]. Any number of instances of a template can be dynamically and independently created. An activity represents a namespace, defining and encapsulating the following elements:

—A fixed set of roles.

—A fixed set of operations associated with each role.

—A set of object types that are created and accessed through the role operations.

—A fixed set of child activity templates that can be instantiated through the execution of role operations. Each nested activity instance defines an independent and separate namespace.

—A dynamic set of events that are generated during the life-cycle of the activity, representing the execution history of the role operations.

In our model an activity has a fixed number of roles within its scope, and the set of operations associated with a role is also fixed. The creation of new nested child activity results in the creation of new set of roles that are visible only in the scope of that child activity. Events in our model are used for enforcing dynamic security requirements and coordination constraints.

4.2 Roles

A role can be viewed as a protection domain with a set of privileges represented by its operations, which perform actions on the objects in the activity's namespace. A role definition involves specification of three aspects: meta-level policies in regard to admission of users to the role, role operations, and conditions under which a role operation can be executed.

The *admission condition* of a role controls user memberships in the role. It is checked only when a user is to be admitted to a role, and it may not hold later when an operation is invoked by that user. The role *activation condition* associated with a role must be true every time a role member invokes a role operation. Role activation condition can be viewed as a common precondition for all operations in the role. A *role validation* condition can be specified for a role to determine when a participant's role membership needs to be revoked. Role admission and activation constraints, operation preconditions, and validation conditions are used for enforcing dynamic security and coordination requirements.

4.3 Role Operations

A role operation may have a *precondition* and an *action*. An operation's precondition must be true to execute its action. The preconditions are expressed using predicates involving events within the activity's namespace. They can also include predicates related to role memberships in the activity. An operation's action can be one of the following: an object method invocation, creation of a new object, or creation of a new nested activity. It is also possible for an operation not to have any action when the operation is provided solely for coordination purposes.

4.4 Events

Events and event counters [Roberts and Verjus 1977] are used in operation preconditions and role constraints for specifying coordination and dynamic security policies. Events correspond to execution of role operations and creation/termination of child activities. Related to each role operation and activity, there are two types of events: *start,* and *finish.* These events are implicitly generated by the runtime system. An event-based predicate is expressed using logical expressions involving event counts and event attributes.

4.5 Shared Objects

Shared objects are represented in our model by their types and method signatures. For an object, access control policies are derived from the various roles' operations involving that object. These are used by the object servers to control access to their objects [Tripathi et al. 2002].

4.6 Nested Activities

An activity can create child activities to perform certain subtasks. A child activity must be defined within the scope of its parent activity. Each child activity defines its own namespace. The nesting of activities results in creation of a hierarchically structured namespace. A nested activity may need to have access to the objects in the scope of its parent activity. For this, objects in the parent activity's namespace can be passed as reference parameters to a child activity. A nested activity definition includes list of the parameter types.

When creating a child activity one may need to assign members to its roles from the participants present in various role of the parent activity. There are two mechanisms to assign members to a role in a child activity. The first mechanism is *static role assignment* or *role reflection*. In role reflection, all members of the specified roles in the parent activity become members of a role in the child activity when that activity is created, subject to the role's admission constraints. Thus, a role in the parent activity is *reflected* into a role in a child activity. Removal of a member from the reflected role (i.e. the role in the parent activity) also implies removal from the role in the child activity. The second mechanism is *dynamic role assignment* in which the participants of the parent activity to be admitted in a child activity's role are specified at the time of activity instantiation.

4.7 Meta Roles: Creator and Owner

In our specification model, associated with every entity – activity, role, and object – there are two system-defined meta roles called *Owner* and *Creator*. These roles are used by the underlying middleware system for administrative purpose.

The user who instantiates an activity or creates an object is the one and the only member of the *Creator* role for that entity. This role membership is implicit and immutable. This role has no permissions associated with it. We call it a *pseudo role*.

An *Owner* role represents meta-level administrative privileges. An activity specification can specify only one of the roles as the *Owner* role of an entity. This results in assignment of entity specific ownership privileges to the role. In the implementation model, there is no concrete representation of the *Owner* roles. We also call it a *pseudo role*.

The members of the role assigned as the *Owner* role of an entity possess the privilege of executing the *reference monitor* for that entity to enforce its policies. They are responsible for correctly managing and enforcing the policies pertaining to that entity. The reference monitor is a *manager* object which is constructed by the underlying system, containing the entity-specific policies derived from the activity specification. For a role manager, the policies are related to the operation preconditions and role admission and activation constraints. An object manager contains policies for dynamic access control. For an activity, the manager contains the policies for creating nested child activities.

5. A SPECIFICATION MODEL FOR DECENTRALIZED CSCW SYSTEMS

An activity is specified in XML, and it is instantiated by a middleware [Tripathi et al. 2002] to generate the runtime environment for the target system. Before realizing a system from its XML specification, its security properties are verified using model checking. We illustrate the specification model and the verification methodology using an example case-study. Here, rather than using XML, we use a notation that is easy to read and conceptually simple to follow.

5.1 Example Case Study

Using Figure 1, we illustrate 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*. In the *Course*, a nested *Examination* activity

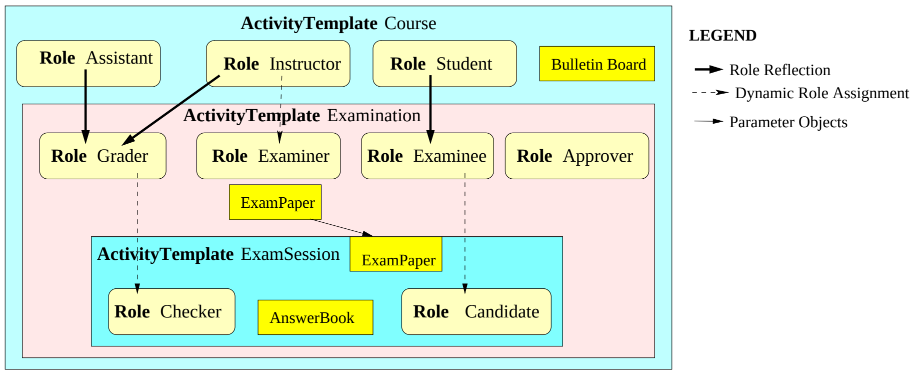

Fig. 1. Role member assignment and object passing in hierarchical structuring of activities

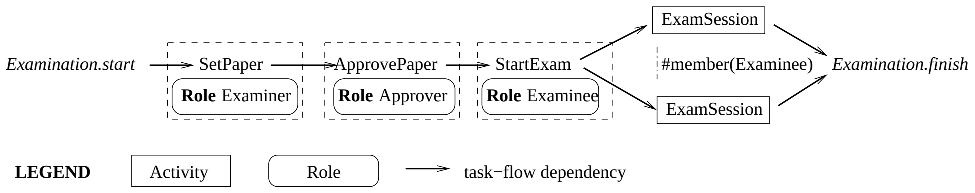

Fig. 2. Task flow requirements in an *Examination* activity

template is defined with four roles: *Grader*, *Examiner*, *Examinee*, and *Approver*. An instance of the *Course* activity template is created for a specific course such as *Chemistry* or *Physics*. Within each such activity instance, any number of *Examination* activity instances may be created, such as *midterm* and *final_exam*. In an examination activity, each member of the *Examinee* role takes the exam by instantiating the nested *ExamSession* activity, which contains the roles: *Candidate* and *Checker*.

The *Instructor* role initiates an *Examination* activity and assigns members to the *Examiner* role. Using *role reflection*, members of the *Instructor* and the *Assistant* roles are admitted to the *Grader* role in an *Examination* activity, and all members of the *Student* role are admitted to the *Examinee* role. Each examinee creates an *ExamSession* activity and he is automatically admitted into the *Candidate* role. A member of the *Grader* role joins the *Checker* role after an exam-session instance is created.

Within an *Examination* activity, there are several tasks that are performed by role members. For example, a member of the *Examiner* role sets the exam-paper, an *Approver* role member approves it, and the members of the *Examinee* role take exam by creating instances of *ExamSession*. These tasks are represented as role operations and nested activities in the specification model as illustrated in Figure 2. The arrows in this figure show the dependency among these operations and activities. For example, the *Approver* role can approve an exam-paper only after the *Examiner* role sets the paper, an *Examinee* role member can start an exam-session only after the *ApprovePaper* operation, and an examination terminates when

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

Fig. 3. Syntax for activity template definition

```
1   ActivityTemplate Course AssignedRoles Assistant, Instructor, Student, Adm2 {
2     Role Assistant{....}
3     Role Instructor {....}
4     Role Student {....}
5       ActivityTemplate Examination Owner Instructor AssignedRoles Examiner, Adm2 {
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  }   }   }
```

Fig. 4. Skeleton specification of Course activity template

the exam-sessions of all of the examinees terminate.

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.

5.2 Activity Template Specification

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. An activity template can specify owner assignment, parameter objects and their types as Java classes, and a termination condition. Moreover, the declaration may list some of the roles that must be assigned members when the activity is instantiated.

In Figure 4, a partial specification of the *Course* activity template of Figure 1 is presented. The activity templates for *Examination* and *ExamSession* are presented in Figure 9 and Figure 10, respectively, and discussed in the following section to illustrate specification of various coordination and security requirements.

In the specification model, the user executing an operation is specified by the pseudo variable *thisUser*. Within an activity, one can refer to its current instance using *thisActivity* and its parent activity instance by *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 it by *thisRole*.

5.3 Condition Specification

There are three kinds of of conditions in the specification model: role membership related, event history based, and temporal, as defined in Figure 5. For temporal condition specification, we use a function *time* that returns the current time.

$\mathit{Condition} \longrightarrow \mathit{RoleCondition} \mid \mathit{OperationCondition} \mid \mathit{TemporalCondition}$
$\qquad\qquad \mid \mathit{Condition}\ \mathit{LogicOp}\ \mathit{Condition} \mid\ !\mathit{Condition}$
$\mathit{RoleCondition} \longrightarrow \#\mathit{RoleMemberList}\ \mathit{Relation}\ \mathit{Count} \mid \text{member}(\mathit{userId},\ \mathit{roleId})$
$\mathit{RoleMemberList} \longrightarrow \text{members(roleId)} \mid \mathit{RoleMemberList}\ \mathit{SetOp}\ \mathit{RoleMemberList}$
$\mathit{TemporalCondition} \longrightarrow \text{time}\ \mathit{Relation}\ \mathit{String}$
$\mathit{SetOp} \longrightarrow \cap \mid \cup \mid \setminus$ $\qquad$ $\mathit{LogicOp} \longrightarrow \wedge \mid \vee$
$\mathit{Relation} \longrightarrow\ > \mid < \mid = \mid <= \mid >= \mid \neq$ $\qquad$ $\mathit{String} \longrightarrow \{\ \text{XML CDATA}\ \}$

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

5.3.1 *Role Membership Functions.* A boolean function `member(thisUser, roleId)` checks if the user executing this function is present in the given role; the role member list is given by the function `members(roleId)`. 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 given by `#(members(roleId))`.

5.3.2 *Event Based Predicates.* The *start* and *finish* events for role operations and activities are implicitly generated by the runtime environment. When an operation is invoked and the operation's precondition is satisfied, the operation's *start* event is generated and the execution of the action part of the operation begins. The precondition-check for an operation and the generation of the corresponding *start* event is atomic. An operation's *finish* event is generated at the end of the operation's actions.

Multiple occurrences of a given event type, such as the *start* events for multiple executions of an operation, are represented by a list. A *list* operator, `( )`, represents the sequence of all 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]`, represents the i'th element in the history of the specified event type. The variables *first* and *last* are used to index the oldest and the most recent elements, respectively, in an event list.

$\mathit{OperationCondition} \longrightarrow \mathit{EventCount}\ \mathit{Relation}\ \mathit{Count}$
$\qquad\qquad \mid \mathit{EventName}\ \text{`['}\mathit{Index}\text{`]'`.'}\mathit{AttributeName}\ \mathit{Relation}\ \mathit{AttributeValue}$
$\mathit{EventCount} \longrightarrow \#\mathit{eventName}\ [\mathit{AttributeName}\ \mathit{Relation}\ \mathit{AttributeValue}]$
$\qquad\qquad \mid \mathit{EventCount}\ \mathit{IntegerOp}\ \mathit{EventCount} \mid \mathit{EventCount}\ \mathit{IntegerOp}\ \mathit{Count}$
$\mathit{EventName} \longrightarrow \text{opId.start} \mid \text{opId.finish} \mid \text{activityId.start} \mid \text{activityId.finish}$
$\mathit{Index} \longrightarrow \mathit{Count} \mid \mathit{EventCount} \mid \texttt{first} \mid \texttt{last}$
$\mathit{AttributeName} \longrightarrow \text{invoker} \mid \text{time} \mid \mathit{String}$ $\qquad$ $\mathit{AttributeValue} \longrightarrow \text{thisUser} \mid \mathit{String}$
$\mathit{IntegerOp} \longrightarrow + \mid \texttt{-} \mid \texttt{mod} \mid \texttt{div} \mid *$ $\qquad$ $\mathit{Count} \longrightarrow \mathit{Integer}$

Fig. 6. Syntax for condition definition: event based predicates

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

Fig. 7. Syntax for role definition

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} }

Fig. 8. Syntax for role operation definition

For each event, there are two predefined attributes: *invoker* and *time*. A subset of an event-list can be derived by filtering it 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 invoked this operation.

Event-based predicates are expressed in two ways: count based or attribute based, 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 initiated the last *opId* invocation is not the same as the current invoker.

5.4 Role Specification

A role specification, as shown in Figure 7, contains the role name, specification for the operations within the role, and three types of role constraints: role admission, validation, and activation constraints. Optionally, it can specify the name of another role that is to be given *owner* privileges for this role, and it can also specify the roles *reflected* into this role. The structure for role operation definition is shown in Figure 8. In the following subsections we illustrate how various kinds of dynamic security requirements can be expressed through role constraints and operation preconditions.

5.4.1 *Role Admission on Activity Creation.* The specification model provides two mechanisms for assigning members to roles. First, using the *Reflect* tag, members of the roles in the parent activity are statically assigned to a role in a *child* activity. Figure 4 presents a partial specification of the *Course* activity. Lines 8 and 9 in Figure 4 show assignment of members to the *Examinee* and *Grader* roles in the *Examination* activity 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.

5.4.2 *Role Admission Constraints.* These constraints control a user's admission to the role to enforce 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.

—An admission constraint specifying that the member count must be less than one to admit a new member in this role:
```
#members(thisRole) < 1
```
—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* constraint 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)
```

5.4.3 *Role Validation Condition.* The validation condition of a role is used to check if a participant's membership in the role needs to be revoked. It is evaluated whenever a role membership query is executed. Figure 9 illustrates use of role validation constraints in the specification of roles in the *Examination* activity. In this example, *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. In Figure 9 (lines 10 and 19), the *Approver* and the *Grader* roles have validation constraints. The validation constraint 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.

5.4.4 *Operation Specification.* As shown in Figure 8, an operation specification includes a name, and may include a *precondition* and an *action.* The operation preconditions allow one to specify coordination constraints and dynamic security requirements. The action part of an operation can create a new object or a nested activity, invoke a method on an object, change ownership of an object, or it can be empty.

The keyword *new* is reserved for specifying creation of an object or an activity. Roles can create only predefined types of objects, specified with a codebase, as defined with *NewObjectDef* in Figure 8.

```
1   ActivityTemplate Examination Owner Instructor AssignedRoles Examiner, Adm2 {
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  }
```

Fig. 9. Specification of *Examination* activity template

```
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 }
```

Fig. 10. Specification of *ExamSession* activity template

The operation dependency requirements expressed in Figure 2 are enforced by the preconditions role operations in the *Examination* activity. In lines 5-7 of Figure 9, the *Examiner* role can perform the *SetPaper* operation only once as specified by the operation precondition. This operation results in the creation of an *exam* object of type *ExamPaper* and granting the operation invoker the *setQuestions* privilege on the object.

Preconditions also facilitate specification of coordination constraints, for both *inter-role* and *intra-role* coordination. For example, in Figure 9 (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 allows each member in the *Examinee* role to independently start an exam session. This illustrates an intra-role coordination policy of *independent participation* by the members in the *Examinee* role.

In Figure 9 (line 6), the precondition of the *SetPaper* operation in the *Examiner* specifies that any one of the role members can execute the *SetPaper* operation. This illustrates *intra-role* coordination based on *cooperative participation.*

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 9 (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.

An activity template specifies the roles that must be assigned members at the time of its instantiation. In Figure 9 (lines 16-17), when an examinee invokes the *StartExam* operation, an instance of the *ExamSession* activity is created, and the participant creating the instance is dynamically assigned to the *Candidate* role. It also passes the *exam* object as a parameter to this activity.

5.4.5 *Role Activation Constraints.* This constraint for a role specifies the common preconditions for all operations defined for that 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 the least 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* roles must be present during the role operations.

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

5.5 Meta Policy Specification

The rules for *Owner* assignment for an entity – activity, role, and object – are as follows:

(1) *Static Ownership Assignment:* The template specification may indicate which role would be the owner of an entity. The creator of entity can be specified as its 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 exists among owners. For an object, a role defined in the encapsulating activity, or in any of its ancestor activities, can be specified as its owner.

(2) *Default Ownership Assignment:* 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.

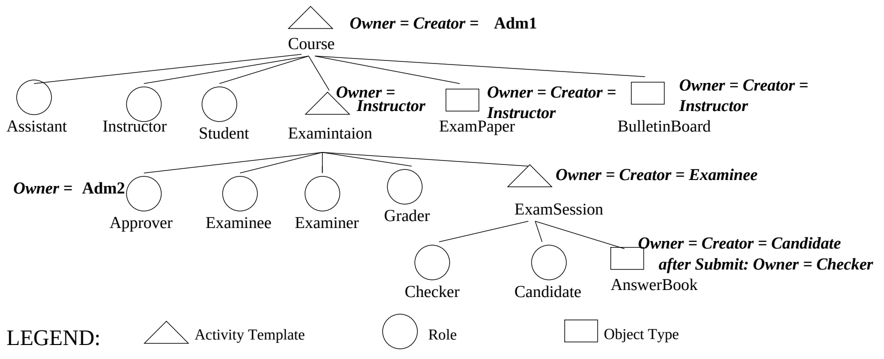

Fig. 11. Owner-assignments in the nested *Course* activity template specification

(3) *Dynamic Ownership Assignment:* 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 creates 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.

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 the *Course* activity example, a similar requirement is specified, which requires that the *Approver* role must be managed by a role in an outside organization.

Suppose that *Adm1* represents the *Creator* of an instance of the *Course* activity. In Figure 4, role *Adm2* is specified as a parameter for this activity. When instantiating this activity, it may be specified as a role in some outside organization. This role is assigned as the *Owner* of the *Approver* role in an instance of the *Examination* activity.

Figure 11 shows the specification of the owners for the entities nested in a *Course* activity template. Figure 12 presents the resulting ownership relations among the entities in Figure 11, based on the given specification and the default ownership rules.

In Figure 11, by default rules, as the *Adm1* role is the creator, it is the owner of the top level *Course* activity instance. 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

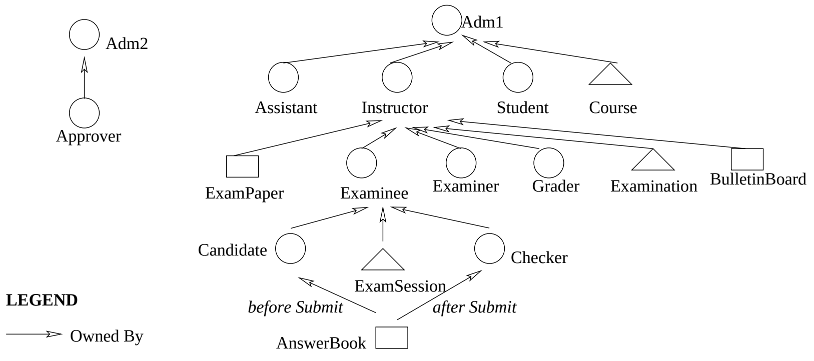

Fig. 12. Owner hierarchies derived from Figure 11

ownership relations form hierarchical structures. There are two owner-hierarchies, under *Adm1* and *Adm2*, because these two roles are from two different organizations and do not have any common ancestor role in the owner hierarchy.

6. DESIGN VERIFICATION GOALS IN MODEL CHECKING

In this section we present different aspects of coordination and security requirements that a collaboration designer may specify as properties to be verified during the design process.

6.1 Verification Properties

Verification of a CSCW design has two distinct goals. First, it has to ensure that the design specification is not inconsistent. Second, it has to ensure that security and coordination requirements are satisfied by a specified design.

6.1.1 *Inconsistent Specification.* Due to incorrect operation preconditions and role membership constraints, an operation can never be executed or a role can never have a member. Such incorrect specifications result from inconsistent requirements or wrong specification of requirements. These incorrect specifications relate to the following two types of properties in our model:

(1) Reachability of Operations: A primary correctness requirement is related to liveness properties that each of the role operations can be executed, i.e., all operations 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** #(Op2.finish) = 1
    Operation Op2 **Precondition** #(Op1.finish) = 1

(2) Satisfiability of Role Membership Constraints: Incorrect or inconsistent specification of role constraints can result in conflicting conditions for admission and validation. 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 re-

quire that its member has to be a member of both role A and B when joining C, which cannot be satisfied.

Role B **Validation Constraints** !member(A)
Role C **Admission Constraints** member(A) $\wedge$ member(B)

6.1.2 *Task-Flow.* The task-flow requirements, i.e, permissible sequence of operations, are specified through preconditions of role operations. In CSCW systems, the collaboration designer may want to verify task-flow requirements independently of other role constraints, with an alternative form of expression. To facilitate such checks during the design, task-flow requirements can be expressed using *path expression* [Campbell and Habermann 1974] constructs, such as `sequence (;)` and `selection (())` with a `count restrictor (:n)`, 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 given below. It 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 the exam-session activity instances has to be equal to the cardinality of the *Examinee* role before the *Examination* terminates.

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

6.1.3 *Role-Based Constraints.* Four types of *separation of duties* constraints – *static*, *dynamic*, *operational*, and *object-based* – and role cardinality constraints can be specified in this specification model. Several role related requirements are specified for the example in Figure 1. To illustrate the verification methodology in the next sections, we choose the following two role constraints (RC) that are representative of such requirements.

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.

6.1.4 *Information Flow and Confidentiality.* 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. Constraints can be specified that certain information can flow to a given role only after some specified conditions are satisfied, or certain information cannot flow to some specific roles. In our case study 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 exam paper before the 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 that candidate's answer book.

6.1.5 *Access Leakage.* In the role-based collaboration model, access rights can only be leaked if unauthorized users can join a role. Unauthorized users may be able to join a role due to incorrect specification of role admission related constraints. 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.

6.2 Verification Problem with Decentralized Policy Enforcement

In a centralized system, with the reference monitor correctly enforcing all specified constraints, the goal of the verification process is essentially to check that the security requirements are not violated due to an incorrect specification. However, in a decentralized CSCW system, where policy enforcement functions are assigned to different participants in the system, the verification goal is also to ensure that a given assignment of owners is *safe*, i.e., it would not result in violation of any sensitive requirements.

In decentralized policy enforcement, when a role is assigned the ownership of an entity, the members of that role are trusted by the designer to correctly enforce the entity-specific policies. Specifically, the owner of a role is trusted with the enforcement of operation preconditions and role membership policies, and an object owner is trusted with the enforcement of object access policies. In this case, there still exists a possibility of security requirement violation due to the *extended* privileges that are acquired by the members in the *Owner* role of an entity. Specifically, these privileges 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. Incorrect assignment of these owner privileges can thus result in violation of confidentiality, information flow, and access leakage constraints.

On the other hand, if all participants cannot be fully trusted for policy enforcement functions, an incorrect ownership assignment may lead to a situation where an "untrusted" participant joins the owner role and may deliberately violate the specified policies for the entity under its ownership. Thus an additional goal of the verification process is to ensure that sensitive security requirements are not violated by untrusted owners.

Consequently, there are two distinctly different assumptions and conditions under which a design can be verified. In the first case, the designer trusts all participants to correctly enforce the security policies for the entities under their ownership control. This means that the specified policies will not be deliberately violated by the owners. We refer to this as the *Verification Model with Trusted Owners*.

In the second case, the designer may trust only a subset of the roles for policy enforcement functions. Thus the verification process is required to ensure that an untrusted participant does not acquire ownership privileges for an entity with some sensitive requirements. This requires the verification model to include the behavior of untrusted participants when they are present in some owner role. We refer to this as the *Verification Model with Untrusted Owners*.

7. VERIFICATION MODEL

Our verification methodology is based on SPIN [Holzmann 2003], which is a model checker with an automata theoretic approach. In SPIN, a model of a system to be verified is specified in PROMELA (a Process Meta Language), which is a C like language with support for inter-process communication primitives. The desired system property can be expressed in LTL (Linear Temporal Logic) using temporal operators `always (□)`, `eventually (◇)`, and `until (U)`. Given the model of a system and a desired property of the system, SPIN converts the model of a system and the negation of a desired system property to finite Buchi automata. Next, SPIN generates a language intersection of these two automata and finds a trace of the counter-example for the desired property.

The well-known challenge in model checking is the state space explosion problem. The search space of the PROMELA model for a small collaboration can be very large. We address here several important issues in applying model checking techniques to our problem domain.

7.1 Model Extraction

In our current work, the collaboration specification in XML is manually converted to PROMELA. Our XML specification only contains the coordination and security properties, thus requiring additional components for runtime control structures to be added to the executable PROMELA specification. In addition to components that manage activities, roles, operations, and events, components are added to the PROMELA specification 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 express various properties in LTL, several primitive predicates are defined. These 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-type, n)*: the number of occurrences of *event-type* is equal to n.

—*access(permission, object_type, user)*: the *user* has the *permission* on an instance of the *object_type*.

In developing the verification models, the search space can be 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 the identities of all the users, but rather maintain a bit variable signifying the fact that some user has invoked the method.

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

```
◇ member(C, Grader)
```

In our implementation, the verification model maintains a bit vector for users, where a bit signifies presence of a user in a role. 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.

$\Diamond$ `member_present[j]`

In SPIN, a variable used in the LTL property to be verified, must be defined in the global scope of the PROMELA specification. Global variables result in a larger number of states. Hence, based on the LTL expression, only the variables that require tracking are defined in the global scope of the PROMELA specification.

7.2 Initial Assignment of Participants

A design is verified with a specified number for initial assignment of participants to various roles. If this number of participants is lower than the number required to verify all the properties, the model checker either provides a trace pointing that the lack of participants resulted in a counter example for a safety property or points the operations that cannot be reached. On the other hand, if the verification is successful with the specified number of participants, it does not ensure that all the verified properties will hold for a larger number of participants. In our research, we have developed a procedure to find a lower bound for the number of initial assignment of participants for a given design. This bound ensures that a larger number of participants will not result in violation of a property that is satisfied with this assignment [Ahmed 2004]. The focus of this paper is on the verification methodology, assuming that an initial assignment of participants is given.

7.3 Aspect-Specific Verification Models

To overcome the state space 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 of 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. We have developed the following five classes of verification models based on the different aspects of the requirements to be checked.

Model for Task-Flow Requirements: It is used for verifying reachability of operations and task-flow constraints, without taking into account the role constraints. It is applicable in cases where the operation precedence constraints do not depend on role membership properties.

Model for Role Constraints: It is used for verifying requirements related to role constraints that do not depend on operation execution history.

Model for Information Flow: This is used to verify properties related to information flow. It is derived by combining some of the aspects of the task-flow and role-constraint models, and it additionally includes control structures to model information flow paths.

Model with Trusted Owners: It is developed by extending the information flow model to verify the safety of the owner assignments in regard to information flow and access leakage due to the "extended privileges" of an owner.

Model with Untrusted Owner: It is used to verify the safety of owner assignments given that a subset of roles may not be trusted for policy enforcement functions. It is derived from the above model with trusted owners.

8. VERIFICATION METHODOLOGY

During the verification process, the designer may find the specification either inconsistent (e.g a specified operation can never be executed or a role can never have a member) or incorrect (e.g. some requirement is violated). In the first case, the design has to be modified, and in the second case either the design or the requirements have to be modified. Due to inter-dependency of the requirements, a modification of the specification may lead to violation of any of the previously verified properties, which would have to be reverified. This can result in a large number of iterations of the verification steps [Kotonya and Sommerville 1998]. To reduce the iterations, our verification methodology follows precedence among the properties it checks. It first checks a design for role and operation related requirements before verifying information flow, access leakage, and ownership related properties. This ordering is motivated by the goal of modeling of primary entities of a specification – activities, roles, and operations – before modifying the design to satisfy higher level security requirements.

The aspect-specific models described above are developed incrementally by adding and removing components that maintain state needed to verify a specific property. In this section, each of the five models is discussed in details including the aspects of a specification that are abstracted in the model and the expressions of the corresponding properties in LTL for verification.

In the first step, the *Task-Flow Model* and the *Role Constraint Model* are applied separately, in any order, for the requirements that are related to the independent aspects of these models. These two models support preliminary verification of task-flow and role constraints that are independent of each other. The requirements that cover the aspects of both these models cannot be verified separately. An example of such a requirement is when admission to a role depends on an operation execution, or when the execution of an operation depends on a role's member count. Such requirements must be verified combining aspects of both these models. Such a combined model also forms the basis for the information flow model.

Next, verification is performed using the *Information Flow Model* to check if any confidentiality properties are violated. It does not consider any extended privileges of the owners. The *Model with Trusted Owners* verifies requirements, such as information flow and access leakage, are not violated due to incorrect assignment of owners. This model is derived from the information flow model by adding appropriate components to represent owners' extended privileges. In the final step, the *Model with Untrusted Owners* is used if any of the roles are designated as untrusted. This model is derived from the trusted owner model by adding components defining the behavior of untrusted owners.

8.1 Verification Model for Task-Flow Requirements

This model is designed to verify aspects related to coordination requirements, such as reachability of operations and task-flow. This model includes components related to activity creation, operations, and preconditions. This model does not include

```
1  proctype ExamSession_Activity( ) {
2    bit Candidate_Write_finish=0, Candidate_OpenExam_start=0, Candidate_OpenExam_finish=0,
3        Candidate_Submit_finish=0, Grader_Grade_finish=0;
4      do
5      ::Grader_Grade_finish == 0 ->
6          if
7          /* Candidate OpenExam */
8          ::  atomic { Candidate_OpenExam_start == 0 -> Candidate_OpenExam_start = 1; }
9                                                    Candidate_OpenExam_finish = 1;
10         /* Candidate Write */
11         ::  Candidate_OpenExam_finish != 0 -> Candidate_Write_finish = 1;
12         /* Candidate Submit */
13         ::  Candidate_Write_finish != 0 -> Candidate_Submit_finish = 1;
14         /* Grader Grade */
15         ::Candidate_Submit_finish == 1 -> Grader_Grade_finish = 1;
16         fi
17     ::  Grader_Grade_finish != 0 -> ExamSession_finish++; break;
18     od }
```

Fig. 13. Task Model in PROMELA for *ExamSession* activity in Figure 9

properties related to users' membership in roles. For any operation preconditions that depend on any role membership constraints, such constraints are assumed to be satisfied. Such requirements are to be verified combining this model with the *Model for Role Constraints*. An exhaustive verification run on this model reports unreachable code, pointing out the operations, which are unreachable.

Figure 13 shows the *Task Model* in PROMELA of the *ExamSession* activity specification, as presented in Figure 10. This model only includes the components that are required to verify operation precedence related properties. In this verification model, each activity is modeled as a process (line 1) and multiple instances of the process can be created. Within such a process, each operation's precondition is modeled as a guarded statement (lines 8, 11, 13, and 15). When the guard becomes true, the statement that follows after the arrow ($->$) is executed in a non-deterministic step. The *atomic* statement (line 8) ensures that the precondition check and generation of corresponding *Candidate_OpenExam_start* event is performed in a single step. The process of the *ExamSession* loops till the termination condition is satisfied (line 17). When the condition is satisfied the global variable *ExamSession_finish* is incremented.

In addition, the path expressions for the task-flow requirements are converted to LTL expressions. In the following, only the response properties of the *Examination* activity, as discussed in Section 6.1.2, are presented in LTL.

```
□( Examination_start → ◇ Examiner_SetPaper_start)
□( Examiner_SetPaper_finish → ◇ Approver_ApprovePaper_start)
□( Approver_ApprovePaper_finish → ◇ ExamSession_start)
□( count(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.

8.2 Verification Model for Role Constraints

This model is developed to ensure that all roles can have members, role membership constraints can be satisfied, and separation-of-duties properties are not violated. It includes only components related to the role membership management aspects, such as static and dynamic role member assignment, role admission and validation constraints.

Within an activity specification, some roles may not have any prerequisite membership constraints for admission. Only the users assigned to these roles can join or be admitted to other roles in that activity. We verify a CSCW design specification based on some given initial assignment of participants to these roles. This initial assignment is important as noted in Section 7.2. In the example *Course* specification, *Student*, *Assistant*, *Instructor*, and *Approver* are initial assignment roles. For the verification process presented in this paper, the initial assignment of participants for a *Course* activity is 5, identified as users A through E in the following assignment: *A* and *B* to *Student*, *C* to *Assistant*, and *D* and *E* to *Approver* and *Instructor* roles. These assignments were determined using the procedure presented in [Ahmed 2004].

Based on the initial members assigned, the model checker reports unreachable code, pointing to the roles that cannot 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.

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

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

```
!◇ ( member(A, Candidate, es1) && !event(ExamSession_start, A, es1))
```

The requirement is specified by negating the fact that eventually user *A* is a member of the *Candidate* role without starting the *ExamSession* instance *es1*. In this expression an activity *es1* is added to imply that the *ExamSession_start* event and the *Candidate* role are in the same activity instance scope. As users *A* and *B* are added to the *Student* role in non-deterministic steps, checking for either of their identities is sufficient for this verification. This requirement was satisfied.

8.3 Verification Model for Information Flow

Several confidentiality properties, such as noninterference, noninference, and nondeducible, have been formalized [Zakinthinos and Lee 1997]. 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` $\longrightarrow$ `o2`.

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

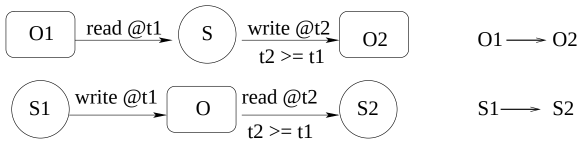

Fig. 14. Information flow: object to object, subject to subject

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 6.1.4, 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,

```
!◇ (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.

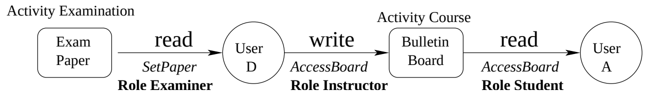

Fig. 15. Trace of a counter-example: *Examiner* leaked *ExamPaper*

• In our initial run, with the assignment of 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*.

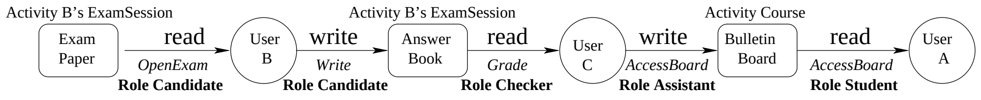

Fig. 16. Trace of a counter-example: *Checker* leaked *ExamPaper*

• In the second run, as shown in Figure 16, candidate B initiated his own *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.

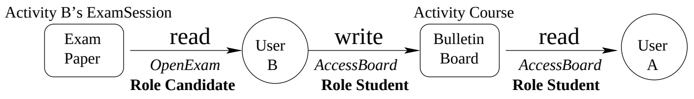

Fig. 17. Trace of a counter-example: *Candidate* leaked *ExamPaper*

• 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 dynamic access control 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(Grader_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.

8.4 Verification Models with Trusted Owners

The primary goal in developing these models is to verify that security requirements cannot be violated due to the extended privileges that are acquired by participants in any owner roles. There are two kinds of models developed in this class: (1) to verify information flow requirements the *Information Flow Model* is extended with owner privileges on role membership information; (2) to verify access leakage properties, the components for information flow are augmented with components representing object access including unrestricted access by owners.
Case Study – Verification of IF1, IF2: In the next step, the information flow properties IF1, IF2 were satisfied with the current owner assignments.
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.

8.5 Verification with Untrusted Owners

The designer designates a subset of the roles that cannot be trusted for policy enforcement. The basic problem in verification of a system with some untrusted roles is to ensure that any specified or potential assignments of untrusted roles as owners for some entities are safe, i.e. they would not result in violation of any sensitive security requirements. Once the untrusted roles have been specified, the next step is to find all the other roles that these untrusted participants would be able to join. Among these roles, a subset may be owners of certain entities. Such an entity is called *potentially misbehaving* as it can be owned by an untrusted participant, who can potentially violate policies associated with it. When verified, if this misbehaving entity violates a given security requirement, it is called a *consequently misbehaving* entity. The goal of our verification process is to identify the *consequently misbehaving* subset of the *potentially misbehaving* entities.

We model the following aspects of the potential misbehavior of an entity owned by an untrusted owner:

1. Violation of role constraints: An untrusted owner of a role may not enforce the role admission and validation constraints and it may admit any user into the role. Additionally, for a role membership related query it may return invalid information. These two behaviors are implemented by removing the role constraints for a role thus resulting in admission of all possible participants in a role and generation of all possible invalid query results.

2. Violation of operation preconditions: A misbehaving owner of a role may not enforce the preconditions associated with the role operations, thus resulting in violation of coordination and dynamic access control policies. It may thus influence other entities by manipulating the causal dependency of the policies under its control. If a misbehaving owner is the notifier of coordination events (e.g. *start* or *finish*), it is modeled either as falsely generating such events or omitting the event notifications. These behaviors are implemented by removing operation pre-

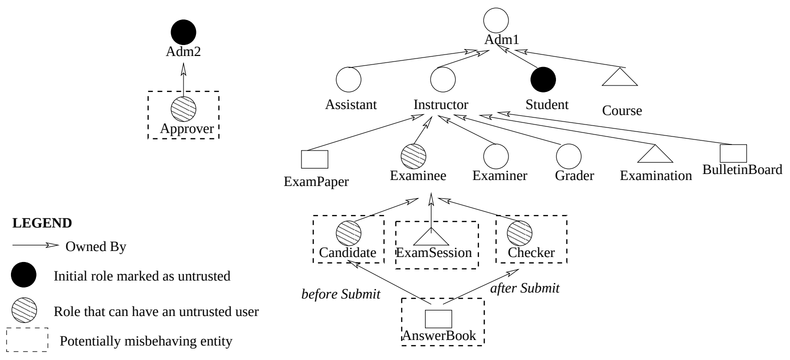

Fig. 18. Potentially misbehaving entities derived based on a trust assignment in Figure 12

conditions, and thus resulting in non-deterministic generation of operation related events. For each subscriber of the operation event, an individual event variable is maintained. These variables are updated in non-deterministic steps to model omission of event notifications.

In verifying requirements with this model, the following steps are performed by the designer:

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 the presence of this misbehaving entity results in violation of a sensitive security requirement, it is marked as *consequently misbehaving* entity. It is then either assigned to be managed by a trusted role or the specification is modified to ensure that such a requirement cannot be violated.

Step 4: If the requirement is not violated, this potentially misbehaving entity may violate the requirement in conjunction with some 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. Steps 2, 3, and 4 are repeated until all the potentially misbehaving entities are selected or all the requirements are verified.

A misbehaving role may generate false coordination events by not enforcing its operation preconditions. This can result in incorrect enabling the preconditions of other role operations. Any such resulting violation of requirements can be prevented by adding the precondition of the misbehaving operation as a part of the preconditions for the affected operations. Any violation of requirements resulting from omission of events cannot be corrected by adding additional preconditions. In such cases, we require that the misbehaving entity be managed by a trusted role.

Case Study – Verification with Untrusted Owner: In our case study example, the designer designated members of the *Adm2* and *Student* roles as untrusted for enforcing policies. These untrusted role are shown by black circles 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 the owner privileges assigned to these roles, the potentially misbehaving entities were found to be *Approver*, *ExamSession*, *Candidate*, *Checker*, and *AnswerBook*. Among the entities, the *Checker*, *Candidate*, and *AnswerBook* are defined in the inner most *ExamSession* activity. We chose the *Checker* role as our first potentially misbehaving entity and found that the requirement *RC1* was violated as the role constraints for the *Checker* role were not enforced. Next, the *Candidate* role was selected and the 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* object 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 were re-verified. 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 to ensure that all properties were satisfied.

9. CONCLUSIONS

The work presented in this paper has been driven by the goal of building a programming framework for constructing secure distributed CSCW systems from their high level specification. We have presented here a role based specification model to express dynamic security and coordination requirements, including administrative security requirements, in distributed CSCW systems. We have also developed a methodology, based on finite-state model checking techniques, to verify the correctness and consistency of a design specification for a given set of security and coordination requirements. Based on the different aspects of the requirements to be verified, we have described development of five classes of models to address problems related to state space explosion and inter dependency of the requirements. An important aspect of this methodology is to verify that the ownership privilege assignments in a design do not result in violation of any critical requirements, when some of the roles cannot be trusted to correctly enforce any policy management functions.

REFERENCES

Ahmed, T. 2004. Policy-Based Design of Secure Distributed Collaboration Systems. Ph.D. thesis, University of Minnesota. Available at http://www.cs.umn.edu/Ajanta/publications.html.

Ahmed, T. and Tripathi, A. R. 2003. Static Verification of Security Requirements in Role Based CSCW Systems. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*. ACM, New York, 196–203.

Ahn, G.-J. and Sandhu, R. 2000. Role-based authorization constraints specification. *ACM Transactions on Information and System Security 3,* 4 (November), 207 – 226.

Atluri, V. and Huang, W.-K. 1996. An Authorization Model for Workflows. In *Proceedings* *of the Fourth European Symposium on Research in Computer Security*. Springer-Verlag LNCS Volume 1146, London, UK, 44–64.

Bacon, J., Moody, K., and Yao, W. 2002. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security 5,* 4 (November), 492 – 540.

Bertino, E., Bonatti, P. A., and Ferrari, E. 2001. TRBAC: A Temporal Role-Based Acces Control Model. *ACM Transactions on Information and System Security 4,* 3 (August), 191 – 223.

Bertino, E., Ferrari, E., and Atluri, V. 1999. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transactions on Information and System Security 2,* 1 (February), 65 – 104.

Bhatti, R., Ghafoor, A., Bertino, E., and Joshi, J. 2005. X-GTRBAC: Am XML-Based Policy Specification Framework and Architecture for Enterprise-Wide Access Acces Control. *ACM Transactions on Information and System Security 8,* 2 (May), 187 – 227.

Campbell, R. H. and Habermann, A. N. 1974. The Specification of Process Synchronization by Path Expressions. In *Operating Systems, International Symposium, Rocquencourt*. Lecture Notes in Computer Science vol.16, Springer Verlag, London, UK.

Corts, M. and Mishra, P. 1996. DCWPL: a programming language for describing collaborative work. In *Proceedings of CSCW'96*. ACM, New York, 21 – 29.

Crampton, J. 2003. Specifying and Enforcing Constraints in Role-Based Access Control. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*. ACM, New York, 43 – 50.

Crampton, J. 2004. An Algebraic Approach to the Analysis of Constrained Workflow Systems. In *Proceedings of 3rd Workshop on Foundations of Computer Security*. 61–74.

Crampton, J. and Loizou, G. 2003. Administrative Scope: A Foundation for Role-Based Administrative Models. *ACM Transactions on Information and System Security 6,* 2 (May), 201 – 231.

Demurjian, S., Ting, T., and Thuraisingham, B. 1993. User-role based security for collaborative computing environments. *Multimedia Review 4,* 2 (Summer), 40–47.

Eshuis, R. and Wieringa, R. 2002. Verification Support for Workflow Design with UML Activity Graphs. In *Proceedings of International Conference on Software Engineering*. ACM, New York, 166 – 176.

Giuri, L. and Iglio, P. 1997. Role templates for content-based access control. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*. ACM, New York, 153 – 159.

Greif, I. and Sarin, S. 1987. Data sharing in group work. *ACM Transactions on Information Systems 5,* 2, 187–211.

Hansen, F. and Oleshchuk, V. A. 2005. Conformance Checking of RBAC Policy and its Implementation. In *First Information Security Practice and Experience Conference (ISPEC 2005)*. 144–155.

Holzmann, G. J. 2003. *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional, New York.

Huang, W.-K. and Atluri, V. 1999. SecureFlow: A Secure Web-enabled Workflow Management System. In *ACM Workshop on Role-based Access Control*. ACM, New York, 83 – 94.

Jaeger, T. and Tidswell, J. E. 2001. Practical Safety in Flexible Access Control Models. *ACM Transactions on Information and System Security 4,* 2 (May), 158 – 190.

Jajodia, S., Samarati, P., and Subrahmanian, V. S. 1997. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 31 –42.

Janssen, W., Mateescu, R., Mauw, S., and Springintveld, J. 1998. Verifying Business Processes using Spin. In *Proceedings of 4th International SPIN Workshop*.

Koch, M., Mancini, L. V., and Parisi-Presicce, F. 2002. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security 5,* 3 (August), 332 – 365.

Kotonya, G. and Sommerville, I. 1998. *Requirements engineering: processes and techniques*. John-Wiley & Sons, Chichester,New York.

LI, D. AND MUNTZ, R. 1998. COCA: Collaborative Objects Coordination Architecture. In *Proceedings of CSCW'98*. ACM, New York, 179–188.

LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. 2002. Design of a Role-based Trust-management Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 114–130.

LI, N., WINSBOROUGH, W. H., AND MITCHELL, J. 2003. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 123–139.

LUPU, E. C. AND SLOMAN, M. 1997. Reconciling Role-Based Management and Role-Based Access Control. In *ACM Workshop on Role-based Access Control*. ACM, New York, 135–141.

MAGGI, P. AND SISTO, R. 2002. Using SPIN to Verify Security Protocols. In *Proceedings of 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318*. 187–204.

MYERS, A. C. AND LISKOV, B. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9,* 4, 410–442.

NYANCHAMA, M. AND OSBORN, S. 1999. The Role Graph Model and Conflict of Interest. *ACM Transaction on Information System Security 2,* 1 (February), 3–33.

OH, S. AND SANDHU, R. 2002. A Model for Role Administration Using Organization Structure. In *ACM Symposium on Access Control Models and Technologies*. ACM, New York, 155 –162.

OSBORN, S. L. 2002. Information Flow Analysis of an RBAC System. In *ACM Symposium on Access Control Models and Technologies*. ACM, New York, 163 – 168.

REITER, M. AND GONG, L. 1995. Securing Causal Relationships in Distributed Systems. *The Computer Journa 38,* 8, 633–642.

ROBERTS, P. AND VERJUS, J.-P. 1977. Towards Autonomous Descriptions of Synchronization Modules. In *Proceedings of IFIP Congress*. North-Holland, Amsterdam, 981–986.

SAMPEMANE, G., NALDURG, P., AND CAMPBELL, R. H. 2002. Access Control for Active Spaces. In *Proceedings of the 18th Annual Computer Security Applications Conference*. 343–352.

SANDHU, R., BHAMIDIPATI, V., AND MUNAWER, Q. 1999. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security 2,* 1 (February), 105 – 135.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-Based Access Control Models. *IEEE Computer 29,* 2 (February), 38–47.

SANDHU, R., FERRAIOLO, D., AND KUHN, R. 2000. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control*. ACM, New York, 47–63.

SANDHU, R. S. 1988. Transaction control expressions for separation of duties. In *Fourth Annual Computer Security Application Conference*. 282–286.

SIMON, R. AND ZURKO, M. 1997. Separation of duty in role-based environments. In *10th Computer Security Foundations Workshop*. IEEE Computer Society Press, Los Alamitos, CA, 183 –194.

THOMAS, R. K. 1997. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In *ACM Workshop on Role-based Access Control*. ACM, New York, 13 – 19.

TRIPATHI, A., AHMED, T., AND KUMAR, R. 2003. Specification of Secure Distributed Collaboration Systems. In *IEEE International Symposium on Autonomous Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, 149–156.

TRIPATHI, A., AHMED, T., KUMAR, R., AND JAMAN, S. 2002. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proceedings of International Conference on Distributed Computing Systems 2002*. IEEE Computer Society Press, Los Alamitos, CA, 393 – 400.

ZAKINTHINOS, A. AND LEE, E. 1997. A General Theory of Security Properties. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 94 –102.