

Static Verification of Security Requirements in Role Based CSCW Systems *

Tanvir Ahmed and Anand R. Tripathi
Department of Computer Science
University of Minnesota
200 Union Street SE, Minneapolis, MN 55455
{tahmed,tripathi}@cs.umn.edu

ABSTRACT

In this paper, we present static verification of security requirements for CSCW systems using finite-state techniques, i.e., model checking. The coordination and security constraints of CSCW systems are specified using a role based collaboration model. The verification ensures completeness and consistency of the specification given global requirements. We have developed several verification models to check security properties, such as task-flow constraints, information flow or confidentiality, and assignment of administrative privileges. The primary contribution of this paper is a methodology for verification of security requirements during designing collaboration systems.

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

Keywords

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 range from online interactive classroom to traditional office automation workflow. The goal of our research is to statically verify security properties of CSCW systems. This requires a formal specification model for security policies and

* This work was supported by National Science Foundation grants ITR 0082215.

development of a verification model to ensure that the requirements are correctly realized by a given design specification. Security policies are, in practice, concerned with subjects' access to resources under specified conditions[20]. Traditional access control models, such as access matrix or *take-grant* model [23], cannot specify such conditions. Policies related to security attributes – privacy or confidentiality, integrity, and resource access – in CSCW need to handle various context sensitive aspects of the collaboration environment. Such context sensitive security constraints can be based on the coordination state of the cooperating users.

In this paper, we present a methodology for verification of security constraints in CSCW systems. In our earlier work [27, 26], we have developed a role-based model for collaboration specification, integrating security and coordination policies. The focus of this paper is on verification of collaboration specifications using the model checker SPIN [10]. The objectives of the verification is to ensure that the specified constraints do not violate any desired coordination and security properties, such as:

- User interactions follow coordination and task flow requirements;
- Roles do not have conflicting or inconsistent constraints;
- Confidential information cannot flow to unauthorized users;
- Authorized information can be accessed;
- Any temporal or conditional constraints on accessing objects can be satisfied; and
- The classical safety property that no rights can be leaked to unauthorized users.

Based on the fact that the classical safety property of the HRU model [9] is not decidable, later access control models have placed constraints on access control structures to facilitate analysis of safety properties, such as Typed Access Matrix [21]. In other cases, users with administrative rights, e.g., *create-subject*, are trusted for not violating security properties. In RBAC (Role Based Access Control) [22], safety of various role based constraints, such as “separation of duties”, have been analyzed with logical expression using rule-based systems [2] and graphical models [11, 17, 18]. In this paper, we present how finite-state techniques, such as model checking, can be utilized for correctness verification in a role based collaboration model. The primary motivation of our approach is to use existing model checking tools to verify security requirements of CSCW systems during the design phase.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SACMAT'03, June 1–4, 2003, Como, Italy.

Copyright 2003 ACM 1-58113-681-1/03/0006 ...\$5.00.

A problem of automata based verification models is state space explosion, i.e., the exponential increase of state size with addition of new system components. In this paper, we present the techniques required to handle the state space explosion problem. Another challenge is the development of verification models to check security constraints, which may be inter-dependent. Modification of the specification to comply with a requirement may violate any of the previously verified requirements. In this paper, we present a methodology on how different aspects of security requirements, such as task flow, role constraints, information flow, assignment of administrative roles, can be verified. Based on the aspects of these requirements, we have developed four verification models using SPIN. We also discuss how these models are constructed from the specification.

The following section presents the required background of our role-based collaboration model. Section 3 discusses techniques for efficient utilization of SPIN. The aspects of security requirements and the verification models that we have developed are presented in Section 4. Section 5 and Section 6 present the related work and conclusions.

2. BACKGROUND: A ROLE BASED COLLABORATION MODEL

Here, we illustrate our role based collaboration specification model through an example, which has been presented in [26, 27]. Like our work, other RBAC models, such as Task Based Access Control (TBAC) [25], Team Based Access Control (TMAC) [24], role based management [13], and role based active security [1] address issues related to context sensitive access control constraints.

In our model, an activity defines how a group of users cooperate towards some common objectives by performing their individual tasks on a set of shared objects. It represents a protection domain and a scope for the roles, objects, and privileges in a collaboration. Roles within an activity are assigned privileges to perform certain tasks. We term these role specific tasks as *operations*. Associated with each role operation is a precondition and an action. An operation can be executed only when its precondition is true. The action associated with an operation can consist of object method invocations, synchronization actions, activity management actions, or instantiation of new objects and activities. An activity can be structured hierarchically, consisting of multiple nested concurrent activities. An *activity template* specifies a generic collaboration pattern. Any number of instances of a template can be dynamically and concurrently created.

2.1 Activity Specification

Figure 1 shows an example of a course activity. The course activity contains three roles: *Instructor*, *Assistant*, and *Student*. These roles have disjoint members and can *read/write* a *BulletinBoard*. The course activity also has a nested *Examination* activity. Members of the *Instructor* and the *Assistant* roles are admitted to the *Grader* role of an instance of the *Examination* activity. Members of the *Student* role of a course join the *Examinee* role in any nested examination activities. Each examinee creates an instance of the *ExamSession* activity template to take the exam. An *exam-session* activity contains the roles *Candidate* and *Checker*. Only the examinee creating this activity can be admitted to

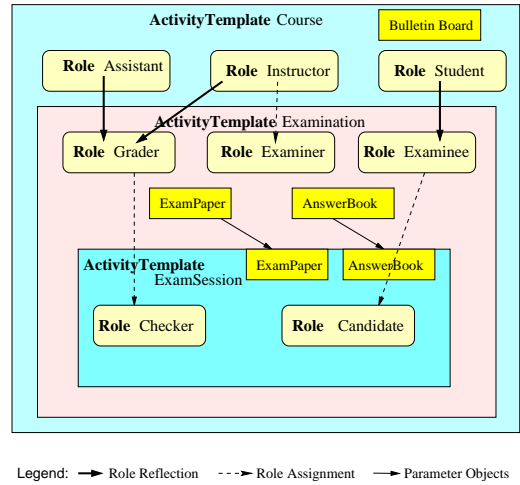


Figure 1: Hierarchical structuring of collaborative activities

the *Candidate* role. References to two objects – *ExamPaper* and *AnswerBook* – are passed as parameters to an exam session activity. A single *ExamPaper* object is shared by all the sessions. On the other hand, a new *AnswerBook* object is created for each exam session.

An *activity template* is specified in terms of roles, operations, and object types. A skeleton of the *Course* activity template, with a complete specification of the nested *ExamSession* activity within the *Examination* activity is shown in Figure 2. In the specification model, within an activity, one can refer to its current instance using the pseudo variable *thisActivity*, and its creator using *parentActivity*. Similarly, pseudo variables *thisRole* and *thisUser* are defined.

2.2 Meta Policy Specification

The *Owner* role specifies which users can manage the activity instance and its nested entities. In this model, the collaboration designer can specify an owner for every entity. The membership rules for the *Owner* role are as follows: (1) The template specification may indicate which role can be the owner of an entity. (2) If not explicitly specified: for an activity, the owner of the parent activity is the owner; for a role, the activity owner becomes its default owner; and the default owner of an object is the role that creates it.

2.3 Event Specification

Events and event counters are used for specifying coordination and dynamic security requirements. Events are implicitly generated by the system, based on the model presented in [19]. Related to each role operation are three types of events: *request*, *start*, and *finish*. For each activity, we have *start* and *finish* events. A count operator $\#$ counts the number of occurrence of a given event type. A subset of the events can be defined based on the event attributes.

2.4 Role Specification

A role specification includes role name, assigned roles if any, role admission and activation constraints, and role operations with their preconditions. In the specification, a boolean function *member(role, user)* checks if a participant is present in a role, and *members(role)* returns the role par-

```

ActivityTemplate Course (AssignedRoles Assistant,
                        Instructor, Student) {
  ObjectType BulletinBoard .....
  .....
  ActivityTemplate Examination (Owner Instructor,
                                AssignedRoles Examiner) {
    ObjectType ExamPaper .....
    ObjectType AnswerBook .....

    Role Examiner {...}
    Role Examinee (Reflect parentActivity.Student) {...}
    Role Grader (Reflect parentActivity.Assistant,
                 parentActivity.Instructor) {...}

    ActivityTemplate ExamSession(Owner Grader,
                                 Objects (ExamPaper exam,
                                           AnswerBook ans),
                                 AssignedRoles Candidate) {
      TerminationCondition #(Checker.Grade.finish)>0
      Role Candidate {
        AdmissionConstraints
          member(thisUser, parentActivity.Examinee)
          & member(thisActivity.Creator, thisUser)
          & #members(thisRole)<1
        ActivationConstraints
          date > DATE(May, 10, 2003, 9:00)
          & date < DATE(May, 10, 2003, 11:00)
        Operation OpenExam{
          Precondition #(OpenExam.start)=0
          Action exam.readPaper()
        Operation Write {
          Precondition #(OpenExam.finish)>0
          Action ans.writeAnswer(data) }
        Operation Submit {
          Precondition Write.finish>0 }
      }
      Role Checker {
        AdmissionConstraints
          #members(thisRole)<1
          & member(thisUser, parentActivity.Grader)
        Operation Grade {
          Precondition #(Candidate.Submit.finish)=1
          Action ans.setGrade(data)}
      }
    }
  }
}

```

Figure 2: Skeleton specification of Course activity template

participant list. The count of the participants in a role is $\#(members(role))$.

2.4.1 Role Admission Constraints:

Role admission precondition must be true when a user is admitted to a role. The *Assistant* role in Figure 1 may have several admission constraints as shown below:

```

#members(thisRole) ≤ 1
& member(thisUser, parentActivity.Staff)
& !member(thisUser, Student)

```

These admission constraints facilitate the specification of: *static separation of duties* constraint requiring that the assistant can not be a student in the course; maximum cardinality constraint of one; *previous qualification* ensuring that the assistant is also a member of the *Staff* role.

2.4.2 Role Admission on Activity Creation:

Our specification model supports two mechanisms to admit participants in a parent activity role to a child activity role. One mechanism is *static role assignment*, also referred

to as *Role Reflection*, in which, using the *Reflect* construct, all members of specified roles in the parent activity become members of a role in the child activity. In Figure 2, using this mechanism, all members of the *Student* role in the parent activity are admitted to the *Examinee* role, subject to the *Examinee* role's admission constraints. Removal of a participant 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 role admission is specified at the time of activity instantiation. The template definition uses the *AssignedRoles* tag to specify which roles must be dynamically assigned at instantiation, as specified for the *Candidate* role in Figure 2. Below is the partial specification of the *StartExam* operation of the *Examinee* role in the *Examination* activity template.

```

Role Examinee (Reflect parentActivity.Student) {
  Operation StartExam {
    .....
    Action {ans=new Object(AnswerBook);
            act=new Activity ExamSession(
                (exam,ans), Candidate=thisUser)}}}

```

When an examinee invokes this operation, an instance of the *ExamSession* template is created and the participant creating the instance is dynamically assigned to the *Candidate* role.

2.4.3 Operation Specification:

Preconditions enable us to specify coordination and dynamic access control policies. Follows an example of an operation specification of the *Examiner* role in the *Examination* activity. The operation *SetPaper* can be performed only once as specified by the precondition. This operation results in creation of an object *exam* of type *ExamPaper* and an invocation of the *setQuestions* method of this object.

```

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

```

2.4.4 Role Validation and Activation Constraints:

Role validation constraints specify the other roles in which a participant should or should not be present for that participant's membership in this role to be valid. Lastly, *activation constraints* for a role specify the conditions that are common preconditions for all operations defined for the role. Every time a user performs a role operation the activation constraints and validation constraints are checked. Various dynamic security policies can be expressed by means of these constraints and preconditions. Figure 2 shows an example of an activation constraint, where the candidate in the exam session activity can perform an operation only during the designated time for the exam.

3. MODEL CHECKING WITH SPIN

SPIN performs exhaustive search to verify safety properties, such as assertions and invalid end-states, and liveness properties related to execution cycles and sequences, such as deadlocks, livelocks, and unreachable code. SPIN specification is written in PROMELA, which focuses on the coordination, i.e., interaction and synchronization aspects of concurrent systems. Given the model of a system and a desired

property of the system, SPIN converts the finite state model of the system and negation of the desired property to Buchi automata and checks if the language intersection of these two automata is empty, which signifies that the property is valid. If the intersection is not empty, SPIN provides a trace of the counter-example. The desired system property can be expressed in LTL using temporal operator **always** (\square), **eventually** (\diamond), and **until** (U).

A system model with all its properties intact produces a large search space. Some of the properties are not in concern when verifying a specific property or can be independently verified. For example, to verify users' admission to roles, modeling of role operations that cannot affect users movement among roles is not required. Properties related to such role operations can be viewed as independent aspects and verified separately. Also, the model can be composed incrementally, adding a new property to the model given a set of properties that have been verified.

The search space of the verification models are reduced by tailoring property specific information. For example, if verification of a property is related to any user's invocation of a method, it is not required for the model to maintain all the users' identities, but rather maintain a bit variable signifying the fact that the user has invoked the method. Verification based on a specific user often can be generalized to verification of global properties. Abstraction of internal data structure also reduces state space. E.g, for a given property interested in the outcome of a precondition, the model does not need to maintain the count of the events in the precondition. Rather a bit is maintained signifying that the precondition is satisfied. Moreover, SPIN supports static analysis, such as slicing algorithms, which provide hints for possible reduction of a model for a given property.

4. VERIFICATION METHODOLOGY

Here we present the security requirements that a collaboration designer may specify as global properties during designing a collaboration specification. We then present a methodology to verify such properties using SPIN.

4.1 Verification of Global Properties

Before verifying global security requirements, verification of the collaboration specification has to ensure that the specification is not incomplete or inconsistent. Our role based collaboration model supports context sensitive and condition dependent access rights. However, the collaboration designer can easily make mistakes when specifying such policies, e.g., a precondition may be incomplete or role admission constraints may be conflicting. Correctness and consistency of task and role modeling and their corresponding specification are often weaved with verification of global security requirements. Here, we discuss the properties that a designer needs to verify for a secure CSCW system. In the next section, we will show how such inter-weaved and multi-dimensional verification concerns can be managed.

4.1.1 Reachability of Operations

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

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

4.1.2 Task Flow

Task modeling is an integral part of collaboration specification in our role model. Though event count based conditions provide expressive power for specifying coordination requirements, the collaboration designer may like to verify task-flow requirement independent of other role constraints, with an alternative form of expression. To facilitate such checks during design, task flow requirements can be expressed using *path expression* [3] constructs, such as **sequence** ($;$) and **selection** ($()$) with a **count** ($:n$) restrictor, where n can be one or more ($+$) or unbounded ($*$). The task flow requirement for the *Examination* activity is expressed as below, which requires that a *SetPaper* is performed before any *ExamSession* can be started, and the number of exam-sessions has to be equal to the cardinality of the *Examinee* role before the *Examination* terminates. Second path, for the *ExamSession*, specifies that the *Write* can be performed one or more time only after completion of an *OpenExam* by the *Candidate* role. After completion of a single *Write*, the *Candidate* can *Submit*. And after the *Submit*, the *Checker* can grade completing the *ExamSession* activity.

```
Examination := Examiner.SetPaper;
              Student.ExamSession:#member(Examinee)
ExamSession := Candidate.OpenExam; Candidate.Write+;
              Candidate.Submit;Checker.Grade
```

4.1.3 Role Based Constraints

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

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

4.1.4 Information Flow and Confidentiality

As RBAC is "policy neutral", we can model information flow constraints by classifying roles with disjoint members with implicit labels. By doing so, a collaboration designer may like to verify if such constraints can be satisfied. Similarly, constraints can be specified that information can flow only after certain conditions are satisfied, or certain information cannot flow to specific roles. In our course activity example, the designer intends to enforce and verify following confidentiality requirements.

1. A participant of the examinee role cannot access the content of the question paper before start of his/her own exam session.
2. Identity of a candidate should not be known to the assistant, who grades the candidate's answer book, before submission of grades.

4.1.5 Integrity and Access Leakage

Due to owner assignments to roles and objects, participants of the owner roles can get extended privileges. Collaboration designer can express integrity policies as global

properties to check that access rights are not unintentionally leaked during owner assignments. In our example, the collaboration designer specifies the following requirement.

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

4.2 Models for Property Verification

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

A second problem is the inter-weaving aspects of the verification requirements. The verification methodology needs to ensure that it follows a precedence among the properties it checks. Consequently, we have developed, based on the aspects of global requirements to be checked, four models for efficient verification. These verification models are automatically constructed from the collaboration specification. The models are termed *Task Model*, *Role Model*, *Information Flow Model* and *Owner Assignment Model*. These four models, respectively, verify correctness of coordination constraints, role constraints, confidentiality requirements with and without assignment of administrative roles, and integrity requirements when owners are assigned.

```

proctype ExamSession_Activity( ) {
  bit Write_finish = 0, OpenExam_start = 0,
    OpenExam_finish = 0, Submit_finish = 0,
    Grade_finish = 0;
  do
    :: Grade_finish == 0 ->
      if
        /* OpenExam */
        :: atomic { OpenExam_start == 0
          -> OpenExam_start = 1; }
          OpenExam_finish = 1;
        /* Write */
        :: OpenExam_finish != 0 -> Write_finish = 1;
        /* Submit */
        :: Write_finish != 0 -> Submit_finish = 1;
        /* Grade */
        :: Submit_finish == 1 -> Grade_finish = 1;
      fi
    :: Grade_finish != 0 ->
      ExamSession_finish++; break;
  od
}

```

Figure 3: SPIN model for ExamSession activity (Task Model)

4.2.1 Verification Model for Coordination Requirements

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

such as intra-role coordination and “operational separation of duties”.

In Figure 3, the conversion of the *ExamSession* activity to PROMELA is shown. In this verification model, each activity is a process and multiple instances of the process can be created. The process of the *ExamSession* loops till the termination condition is satisfied. Each of the conditions is expressed as a guard. When the termination condition is satisfied the global variable *ExamSession_finish* is incremented.

The path expression for the task-flow requirements are automatically converted to LTL expression based on LTL patterns, which corresponds to path expression constructs. The path expression for the *Examination* activity will be converted to the following LTL expressions, where *count(event, n)* signifies a true value when the number of *event* is equal to *n*.

```

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

```

4.2.2 Verification Model for Role Constraints

Primary goal of *role verification model* is to ensure that all roles can have members, and role related constraints can be satisfied. To check the user-related properties, the presence of users in roles is modeled. For a given set of users for the top level activity, the model exhaustively searches and assigns members to nested roles. As this can result in a very large search space, a minimal number for initial user assignment is calculated based on the cardinality of the roles. Verification with the minimal number of initial users can activate all the roles and trigger all the intra-role coordination constraints. We explain the algorithm for finding a minimal number for the initial assignment using the *Course* example. As all the initial roles in the *Course* activity have disjoint members without any cardinality constraints, each requires at least a member. In the second level *Examination* activity, none of the roles require more than one member to be active. However, the *Examinee* role has a “intra-role coordination” requirement, expressed using *thisUser* in operation precondition of *StartExam*, which specifies that each examinee can start only his/her own *ExamSession*. To facilitate multiple instances of *ExamSession* activity, at least two members for the *Examinee* role are required. In the bottom most *ExamSession* activity level, the *Candidate* and *Checker* role have a maximum cardinality constraint of 1. This results in a minimal number for initial users to be 4 with 2 in *Student* role.

Based on the initial members assigned, the exhaustive search could report unreachable codes, in turn, pointing to the membership constraints, which cannot be satisfied. For a large collaboration, the exhaustive search with a minimal members may not be feasible as role membership related events cannot be confined within an activity boundary. In such cases, role properties can be specified in LTL for faster verification, as it results in searching for only specific counter-examples. In the *Course* activity, with initial assignment of user *C* to *Assistant* role, the correctness requirement that the *Grader* role eventually have a member is expressed as the following expression using LTL.

```

□ ◇ member(C, Grader)

```

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

$\square \diamond \text{member_present}[j]$

To facilitate the designer to express various types of role constraints, conversion functions for role constraints to LTL expressions are developed. For example, a tuple for “static separation of duties” for user C is automatically converted to LTL expression as shown below.

```
StaticSOD( Assistant, Student, C ) :=
  ! $\square$ ( member(C, Assistant) && member(C, Student) )
```

4.2.3 Verification Model for Information Flow

Several confidentiality properties, such as noninterference, noninference, and non-deducible, have been formalized, and a comparative discussion on types of information flows in these models can be found in [28]. In our verification model only explicit information flow [6] is captured, which can be summarized in the following two rules:

1. Given objects o_1 , o_2 and subject s , which has *read* permission on o_1 at time t_1 and *write* permission on o_2 at time t_2 with $t_2 \geq t_1$, then information can flow from o_1 to o_2 , i.e., $o_1 \rightarrow o_2$.
2. Similarly, given o is an object and subject s_1 has *write* permission on o at time t_1 and subject s_2 has *read* permission on o at time t_2 with $t_2 \geq t_1$, then information can flow from s_1 to s_2 , i.e., $s_1 \rightarrow s_2$.

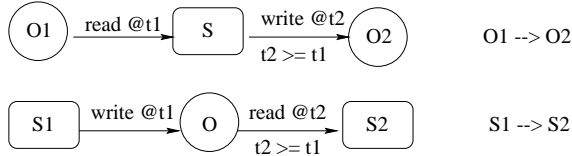


Figure 4: Information flow: object to object, subject to subject

To incorporate the above two rules in the model, components related to users’ knowledge and objects’ internal information are added. In the model, *read* permission is assumed when a method returns any values, and *write* is assumed when any values are passed during method invocation or object creation.

Verification of Requirement 1: Knowing that user A and B are initial members of the *Student* role, we express the information flow requirement 1, in Section 4.1.4, as “user A of the examinee role cannot access the content of the question paper before start of his/her own exam session” as below, where $knowledge(subject, object)$ signifies that the *object* content has passed to the *subject*, and $event(event_type, user)$ signifies that the *user* has triggered the specified type of event.

$! \diamond (\text{knowledge}(A, \text{ExamPaper}) \ \&\& \ !\text{event}(\text{ExamSession_start}, A))$

The requirement is specified by negating the fact that eventually user A knows the content of the *ExamPaper* without starting his/her *ExamSession*. In our initial run, with initial assignment of users A and B to *Student*, C to *Assistant*, and D to *Instructor*, a counter-example was found. In the trace, the *Examiner* leaked the *ExamPaper* to the examinees before start of their exam-sessions through the *BulletinBoard*. To encode that such an act would not be performed by the *Examiner*, we provided the fact to the model as a tuple $!write(Examiner, ExamPaper, BulletinBoard)$, which meant that *Examiner* would not *write ExamPaper* content to the *BulletinBoard*.

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

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

In the first solution, the *Student* role’s privileges on *BulletinBoard* were revoked during the *Examination* activity. This was accomplished by adding a coordination requirement on the operations of the *Student* role accessing the *BulletinBoard*. However, this was very restrictive as the examinee, who had not started an exam-session, was not able to use the bulletin board.

In the second solution, we ensured that the examinee who initiated an exam-session could not access the *BulletinBoard* during the session. To enforce that, we revoked *BulletinBoard* privileges from the *Student* role and added a new role *BulletinBoard-User* in the *Course* activity with *BulletinBoard* privileges. Only by joining this role a participant of the *Student* role could access the *BulletinBoard*. Additionally, an “operational separation of duties” constraint was specified that an examinee had to leave the *BulletinBoard-User* role during his/her *ExamSession* activity. However, the verification of the confidentiality property failed as examinee A was able to wait for completion of examinee B’s exam-session and was able to share the *ExamPaper* through the *BulletinBoard*. To circumvent this, we added a constraint that the exam-session could only be started after a predefined time, *ExamStartTime*. Moreover, the submit of the *AnswerBook* could not be performed for a period *ExamJoinDuration*, during when other examinees could initiate their exam-sessions. To include these restrictions, we required real time events, however, SPIN does not support verification related to real time. As the proper duration of these events were not necessary for verification, we modeled two events, which signified the events *ExamStartTime* and *ExamJoinDuration*. These events were modeled based on the *SetPaper* event and the *Submit* event of the first examinee, respectively. Finally, the resulting specification was valid for the requirement 1.

4.2.4 Verification with Owner Assignment

In CSCW systems, it may not be possible to assign a single administrator for all the collaboration aspects. Moreover, as roles within an activity can be from different administrative

domains, they may need to be maintained by disjoint owners. In our collaboration model, owner of an activity can terminate the activity, owner of a role can admit/remove members from the role subject to role admission/validation conditions, and owner of an object has unlimited access to the object. As owner assignment related concerns can be separately checked, in the previous discussion of information flow model, we have not taken into consideration the effect of assigning owners to different entities. A motivation behind the *owner model* is to ensure that due to owner assignments, no confidentiality requirement is violated or no access right is leaked to unauthorized users.

In modeling owner rights, we assume that the protection mechanism is not tempered by the owner, such as the owner cannot or do not bypass precondition checks. Within the context of collaboration systems, we also assume that the owners are not adversarial as they are participating in cooperative activities. This implies that we trust them to follow policies derived from the specification.

Within this trust framework, the components related to owner rights are modeled. In our *owner assignment model*, the owner of a role can view identities of the participants of the role. Moreover, the owner of an object can read/modify it without any restriction.

Verification of Requirement 1 under Owner Model:

Going back to our verification case study, in the next step, we checked if the confidentiality requirement 1 is preserved with owners. We assigned *Instructor* as the owner of the *Examination* activity and its nested roles, and *Grader* as the owner of *ExamSession* activities and its roles, as shown in Figure 2. This owner assignment configuration satisfied requirement 1.

Verification of Requirement 2: The confidentiality requirement 2, knowing that user *C* is an initial member of *Assistant* role, can be expressed as below. In this expression a context *es1* is added to imply that the *Grade_finish* event and the *Candidate* and *Checker* roles are in the same activity instance scope. Moreover, *identity(user)* represents the information object for user’s identity.

```
!◇( !event(Grade_finish, C, es1) && member(A, Candidate, es1)
&& member(C, Checker, es1) && knowledge(C, identity(A)))
```

The requirement is expressed as a negation of the error behavior, that is *A* is a member *Candidate* role and *C* is a member of the *Checker* role in the same exam-session context, and *A*’s identity is known to *C* before *Grade* is finished by *C*. As user *A* and *B* are added to the *Student* role with non-deterministic steps, checking for either of their identity in *C*’s knowledge suffices for this verification.

We found that this requirement could not be satisfied when the *Grader* was the owner of the *Candidate* role and had access to their identities. Hence, we tried with *Examinee* as the owner of the *Candidate* role. A counter example was found where candidate *A* leaked his/her identity through the *AnswerBook* object. The fact that *Candidate* would not perform such an action was provided to the model. In the consequent verification, requirement 2 as well as 1 were satisfied with *Examinee* as the owner of the *Candidate* role.

Verification of Requirement 3: The requirement 3 is related to access leakage that the *write* privilege to the *An-*

swerBook 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*. With *Examinee* as the owner of the *Candidate* role, the verification of requirement 4 failed. We set *Examinee* as the owner of the *Candidate* role, and all the requirements 1, 2, and 3 were satisfied.

5. RELATED WORK

Model checking is used for various application specific property verification, such as verification of workflow properties [15, 7, 12]. MENTOR[15] utilizes state and activity chart to formally specify business processes and verifies the workflow specification using symbolic model checking based on ordered binary decision diagram. Model checkers, which support LTL property, are used for verification of workflow processes using NuSMV [7] and SPIN [12]. Compared to our work, these systems mainly specify task coordination requirements and do not provide a model to verify security properties such as information flow. Static analysis for controlling information flow using program analysis is addressed by several researchers [16, 6]. These analysis concentrate on program verification for ensuring multi-level security policies related to information flow through storage channels.

Finite state based model checking using tools, such as SPIN, is also explored for program analysis [5, 6] and protocol verification [14]. In this paper, we present a role based model to specify both coordination and security requirements and utilize finite state models for verification of the specification. An approach for specification of functional and confidentiality requirements in CSCW systems using *Z* notation for formal verification is presented in [8]. In [4], the use of a Petri-Net tool for validation of system security policies, specifically mandatory access policies, is proposed. In contrast, we present a methodology for specifying and verifying security constraints in role based CSCW systems using finite-state based model checking.

6. CONCLUSION

In this paper, we have presented a methodology for verification of security requirements of CSCW systems. In our previous work, we have developed a role based model for specifying collaboration systems. In this paper, we presented how such specification can be statically verified with finite-state techniques using SPIN. The global requirements to be verified are converted to LTL expressions and are checked against SPIN models. We have presented techniques to handle the state space explosion problem of finite-state based models, developing incremental modeling with separation of concerns and property specific abstractions. Based on the types of security requirements in CSCW systems, such as task-flow constraints, role based constraints, information flow and confidentiality, and management of administrative roles, we have developed four different verification models. Using a case study of a course activity, we have shown how the global requirements can be specified and verified with the models in SPIN. In the current model, we are only concerned with explicit information flow. In our future

work, we would like to verify other types of confidentiality properties [28].

7. REFERENCES

- [1] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492 – 540, November 2002.
- [2] E. Bertino, E. Ferrari, and V. Atluri. A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. In *ACM Workshop on Role-based Access Control*, pages 1–12, 1997.
- [3] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In *Operating Systems, International Symposium, Rocquencourt*. Lecture Notes in Computer Science vol.16, Springer Verlag, April 1974.
- [4] S. Castano, P. Samarati, and C. Villa. Verifying System Security Using Petri Nets. In *Security Technology, 1993. Security Technology, Proceedings. Institute of Electrical and Electronics Engineers 1993 International Carnahan Conference on*, pages 244–250, Oct 1993.
- [5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proc. of ICSE'2000*, pages 439 – 448, 2000.
- [6] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
- [7] R. Eshuis and R. Wieringa. Verification Support for Workflow Design with UML Activity Graphs. In *Proc. of ICSE'2002*, pages 166 – 176, 2002.
- [8] S. Foley and J. Jacob. Specifying Security for Computer Supported Collaborative Computing. *Journal of Computer Security*, 3(4):233–253, 1995.
- [9] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461 – 471, August 1976.
- [10] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [11] T. Jaeger and J. E. Tidswell. Practical Safety in Flexible Access Control Models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):158 – 190, May 2001.
- [12] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying Business Processes using Spin. In *Proc. of 4th International SPIN Workshop*, 1998.
- [13] E. C. Lupu and M. Sloman. Reconciling Role-Based Management and Role-Based Access Control. In *ACM Workshop on Role-based Access Control*, pages 135–141, 1997.
- [14] P. Maggi and R. Sisto. Using SPIN to Verify Security Protocols. In *Proc. of 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318*, pages 187–204, 2002.
- [15] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz Dittrich. *Enterprise-wide Workflow Management based on State and Activity Charts, Workflow Management Systems and Interoperability in: A. Dogac, L. Kalinichenko, T. Ozsu, A. Sheth (Eds.): Springer Verlag*, 1998.
- [16] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129 – 142, 1997.
- [17] M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Transaction on Information System Security*, 2(1):3–33, February 1999.
- [18] S. L. Osborn. Information Flow Analysis of an RBAC System. In *ACM Symposium on Access Control Models and Technologies*, pages 163 – 168, 2002.
- [19] P. Roberts and J.-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proc. of IFIP Congress*, pages 981–986, 1977.
- [20] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference, who needs it? In *14th IEEE Computer Security Foundations Workshop*, pages 237 –238, 2001.
- [21] R. Sandhu. The Typed Access Matrix Model. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 122 –136, 1992.
- [22] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.
- [23] L. Snyder. Formal Models of Capability-Based Protection Systems. *IEEE Transactions on Computers*, C-30(3):172 – 181, March 1981.
- [24] R. K. Thomas. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In *ACM Workshop on Role-based Access Control*, pages 13 – 19, 1997.
- [25] R. K. Thomas and R. S. Sandhu. Conceptual Foundations for a Model of Task-based Authorizations. In *IEEE Computer Security Foundations Workshop*, pages 66–79, 1994.
- [26] A. Tripathi, T. Ahmed, and R. Kumar. Specification of Secure Distributed Collaboration Systems. In *IEEE International Symposium on Autonomous Distributed Systems (ISADS)*, April 2003.
- [27] A. Tripathi, T. Ahmed, R. Kumar, and S. Jaman. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proc. of International Conference on Distributed Computing Systems 2002*, pages 393 – 400, July 2002.
- [28] A. Zakinthinos and E. Lee. A General Theory of Security Properties. In *IEEE Symposium on Security and Privacy*, pages 94 –102, 1997.