

Secure Management of Distributed Collaboration Systems *

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

Abstract

Our goal is to construct secure distributed collaboration systems by coupling their high-level specifications with a policy-driven middleware. This paper addresses three problems related to decentralized management of collaboration systems by a middleware. To correctly enforce security and coordination policies, the middleware determines which nodes can be trusted for enforcing policies related to different aspects of a collaboration system. Secondly, the middleware ensures consistency in evaluation of coordination constraints. Lastly, security and integrity of event communication is addressed. Moreover, we present the steps for building collaboration systems using our middleware.

Keywords: Distributed collaboration, CSCW, Middleware, Process coordination, Role based access control, Security policy specification.

1 Introduction

The goal of our work is to realize distributed collaboration systems from their high level specification capturing coordination and security policies. In this paper, we address the problems that arise in enforcing these policies in decentralized management of distributed collaboration systems. Policies are used to manage various dynamic aspects of collaboration systems: new users, whose identity may not be known a priori, may join the collaboration; new artifacts may be introduced at runtime; or cooperating participants may need to change their coordination policies. Other than coordination constraints, a collaboration specification in our model also captures various security constraints, such as “separation of duties”, “history based constraints”, and dynamic or context-sensitive access control policies [12]. In collaboration systems, coordination constraints are often weaved with access control concerns, which is also evident in task based access control (TBAC) [9]. For example, in

an examination activity, students can only view the question after the examiner has released it and only during the exam-session. Although this resembles a coordination concern, it is also an access control constraint.

In Internet-wide collaboration, users and shared objects are inherently distributed, and users from different organizations or administrative domains need to cooperate. There are several motivations for decentralized management of collaboration systems. Other than the well-known motivations of scalability and elimination of “single point of attack”, collaboration systems need decentralized management as no single organization, site, or user could be trusted with all the management aspects of these systems. In our collaboration model, all users are not equally trusted. Even within the scope of a collaboration activity, participating users may compete to maximize their personal gains. A user can only be trusted to maintain integrity of collaboration entities, i.e., properly enforce policies associated with the entities, if he/she has vested interest to do so.

We have developed a role-based collaboration specification model and have implemented a prototype middleware [11]. From the specification of a collaboration, policy modules for coordination, security, and management aspects of the collaboration are derived. The middleware automates the realization of the collaboration system by integrating these policy modules with generic managers. In this paper, we present the solutions that we have developed and incorporated in the middleware to address the following security problems that arise in decentralized management of collaboration systems.

1. Not being able to trust any single node for all management tasks imposes constraints on the middleware in supporting the security requirements of access control, authentication, confidentiality, and non-repudiation. Mechanisms are required to find appropriate nodes, which can be trusted to enforce various security policies derived from the collaboration specification.
2. In decentralized management of a collaboration, correct enforcement of coordination constraints is an important problem as the system state is distributed and

* This work was supported by National Science Foundation grants ITR 0082215 and EIA 9818338.

a node’s view of it may be incomplete or stale. As dynamic access control policies are closely tied to coordination constraints, security of the system requires consistency in evaluation of such constraints.

3. Security of event communication requires that events are not tampered with, are generated by correct notifiers, and only received by authorized subscribers. Importantly, when roles or shared objects are maintained in different trust domains, the middleware needs to ensure that events are not *falsified* or *omitted*, i.e., the occurrence of an event can be trusted.

Compared to our work, only a few systems, such as COCA [4] and DCWPL [1], have taken the approach of building a collaboration environment by coupling its high level specifications with a middleware. In contrast to our work, coordination rather than security is the main concern of these systems, and at the implementation level most of these systems use centralized solutions.

The following section presents an overview of our role based collaboration model. In Section 3, we present a trust model to select nodes for secure decentralized management of a collaboration system. Section 4 presents rules for ensuring correctness in decentralized enforcement of coordination constraints. Section 5 discusses the trust and integrity issues in event communication. Section 6 discusses the steps in constructing a collaboration system from its specification. Sections 7 and 8 present the related work and conclusions.

2 Role Based Collaboration Model

Here we give a brief overview of our role based collaboration model [10].

Roles: In our collaboration model, users are represented by their roles, and roles are assigned privileges to perform certain tasks. We term these role specific tasks as *operations*. Operations can be method invocations on shared objects, synchronization actions, or management related actions. Role operations may have *preconditions* to coordinate users’ actions.

Users acquire privileges to perform tasks in the collaboration by joining or being admitted to roles. Role *admission constraints* specify the conditions that need to be satisfied when a user joins a role. 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 acti-

vation constraints and validation constraints are checked ¹. These constraints facilitate dynamic access control policies.

Activities: An *activity* is an abstraction of a collaboration session, which provides a protection domain and scope for the roles, objects, and privileges in the collaboration. An activity can be structured hierarchically, consisting of multiple nested concurrent activities. Objects can be passed into nested activities and users in roles from the parent activity can be statically or dynamically assigned to roles in nested activities. An *activity template* specifies a generic collaboration pattern among a set of roles using some shared objects. Activities are instantiated from templates.

Event-based Specification Model: Operation preconditions and other role related constraints are expressed in terms of event count based predicates [7], role membership predicates, and their connectives. Events correspond to the execution of role operations. Related to each role operation are three types of events: *request*, *start* and *finish* [7]. A count operator # on events returns the number of occurrences of a given event type. An event list can be filtered based on event attributes. For example, the following operation related event is filtered based on the invoker’s identity: `#opName.start(invoker = John)`.

We define role membership functions: *member(user, role)* returns true if the user is a member of the specified role, and *members(role)* returns the list of all the members in that role. The operator # on the list returns membercount. Our specification model uses terms, such as *thisUser*, *thisRole* and *parentActivity*, to facilitate context sensitive policy specification.

In our middleware, events can be subscribed in the form of primitive events, event counts, event based predicates, or role membership predicates. Use of predicates instead of primitive events reduces event communication.

An Example Specification: We present here the example in Figure 1 to give an overview of our specification model. The example is based on the one discussed in [11], to show some additional features of the specification language and to motivate a trust model for decentralized management of entities in a collaboration. In Figure 1, we illustrate a template for an *Examination* activity containing three roles: *Grader*, *Examiner*, and *Student*. “Static separation of duties” constraints specified on these roles ensure that a participant of any of these roles is not a member of the other two roles. Each student creates an instance of a nested *ExamSession* activity template to take the exam. An exam session activity contains two roles: *Candidate* and *Checker*. The *Examination* activity instance has multiple concurrent

¹Role validation constraints and activation constraints do not apply for role management related operations, such as *join* and *leave*.

exam-session activity instances initiated by different participants of the *Student* role. Only the student creating an exam session activity can be admitted to the *Candidate* role. Two members of the *Grader* role can join the *Checker* role to grade that exam session. The *Examiner* role creates the *ExamPaper* object, passing it to nested exam-session activity instances. Moreover, the *Candidate* role creates an *AnswerBook* object in an exam-session. The specification of the nested *ExamSession* activity template is shown in Figure 2.

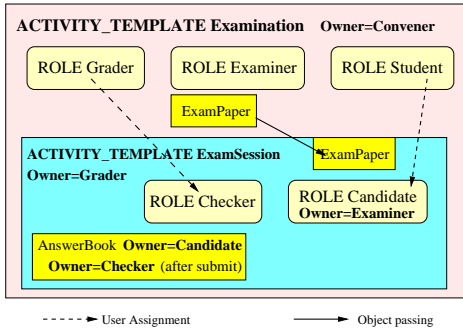


Figure 1. Example of an activity template

```

ActivityTemplate ExamSession(Owner Grader,
                             Objects (Course.ExamPaper exam)) {
  TerminationCondition #(Checker.Grade.finish)>0
  Role Candidate {
    Conceal (participant-id, (Checker))
    AdmissionConstraints
      member(thisUser, parentActivity.Student)
      & member(thisActivity.Creator, thisUser)
      & #members(thisRole)<1
    ActivationConstraints
      date > DATE(Mar, 22, 2002, 8:00)
      & date < DATE(Mar, 22, 2002, 11:00)
    Operation StartExam {
      Precondition #(StartExam.start)=0
      Action { ans=new OBJECT(AnswerBook)
              exam.readPaper()}
    Operation Write {
      Precondition #(StartExam.finish)>0 }
      Action ans.writeAnswer(data) }
    Operation Submit {
      Precondition #(Write.finish)>0
      Action ChangeOwner(ans, Checker) }
  }
  Role Checker {
    AdmissionConstraints
      #members(thisRole)<2
    ValidationConstraints
      member(thisUser, parentActivity.Grader)
    Operation Grade {
      Conceal (participant-id, (Candidate))
      Precondition #(Candidate.Submit.finish)=1
      Action ans.setGrade(data)}
  }
}

```

Figure 2. Specification of ExamSession activity template

3 Trust Model for Secure Management of Collaboration

In a collaboration system, though participants are cooperating, they can not be equally trusted to maintain shared objects or enforce coordination and security policies. Participants may be from different administrative domains, and they may not trust any one particular domain with all the management tasks. In our model, policies for entities – activities, roles, and objects – need to be enforced at trusted nodes. We consider a node to be *trusted* for managing an entity, if the administrator of that node has vested interest in enforcing the collaboration policies specific to that entity.

In our model, a meta-role, *Owner*, for each collaboration entity specifies which users can be trusted to manage that entity. An entity is managed at a server that is trusted by its owner. The owner of an entity possesses certain privileges. The owner of a role can add or remove users from the role. An object’s owner can modify the object’s access control policies.

The requirements that arise in selecting a role as the owner of an entity are discussed below. We relate these requirements with the example in Figure 1 to present the motivations for designating owners in a collaboration system.

1. Generally the role creating a nested activity may be allowed to manage the activity. In some situations, the creator may not be trusted to manage the activity it creates. In our example, a student may not be trusted to enforce various security policies of the exam session activity that he/she creates.
2. It may not always be possible to trust an activity’s owner to manage all the roles within its scope, as participants in these roles may be from domains other than that of the owner.
3. An object within a collaboration may need to be managed by a role which is trusted by all roles sharing the object. In the example, the exam paper object needs to be managed by a role trusted by the *Examiner*, *Candidate*, and *Checker* roles.
4. A single entity may not be trusted to manage an object throughout its life-cycle. In the exam-session activity, an answer book is created and initially managed by the candidate; however, after the submission of the answer book the candidate cannot be trusted to manage it.
5. In our collaboration model, several types of confidentiality constraints for a role’s participant-related attributes can be specified:

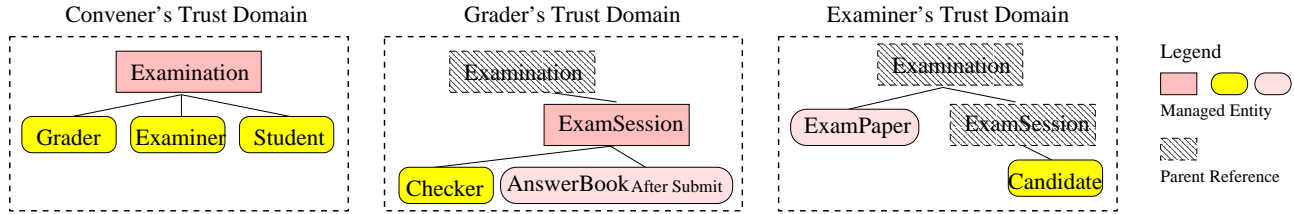


Figure 3. Entities in Figure 1 managed by different owners' trust domains

(a) A role may need to hide its participants' identities from other roles. This implies that events generated by the role cannot contain the participants' identities, and no membership related queries are allowed. In the specification of Figure 2, such a constraint that the *Candidate* role participant's identity should not be known to the *Checker* is specified using the *Conceal* primitive.

(b) In a less restricted situation, a role may allow membership related queries but hide participants' identities in role operations. The specification in Figure 2 provides such a constraint that the *Checker* role must hide the identities of the participants performing the *Grade* operations using the *Conceal* primitive.

3.1 Ownership of Collaboration Entities

Based on the security requirements expressed in a collaboration specification, the middleware derives trust relationships among all the collaboration entities using the following ownership assignment rules. The runtime system assigns a trusted node for managing an entity based on owner assignments.

1. The activity definition, i.e. activity template, may indicate which role is the owner of an entity. The role designated as the owner must exist when the entity is created. (a) For an activity, a role in the scope where the activity is defined or in the outer scope may be designated as its owner. (b) Same rule applies to an object. Moreover, ownership changes can be specified. (c) For a role, the owner must be a role in its outer scope.
2. If the owner of an entity is not specified, the default ownership rules apply. For an activity, the owner of the parent activity becomes its owner. A *Convener* role is defined as the owner of the top level activity instance. The default owner of a role is the owner of the activity in which it exists, and the default owner of an object is the role which creates it.

In our model, the confidentiality requirements have an implication on owner assignments. For example, the confidentiality requirement that the *Checker* role must not know

participants' identities of the *Candidate* role imposes a restriction – any role whose members may be admitted to the *Checker* role, can not be specified as the owner of the *Candidate* role. If this restriction is not imposed the confidentiality requirement can be violated. For example the *Grader* role could be specified as the owner of the *Candidate* role. In that case, a member of the *Checker* role also being a member of the *Grader* role could access the *Candidate* role's membership information.

Figure 3 shows the entities managed by different owners in their trust domains. These owners are designated based on the specification and the ownership assignment rules. For the top level *Examination* activity, the *Convener* role is the owner. By the default rule, the *Convener* is also the owner of the *Grader*, *Examiner*, and *Student* roles. The *Grader* role is specified as the owner for nested *ExamSession* activity instances, satisfying requirement 1. By the default rule, the *Examiner* is the owner of the *ExamPaper* object. This satisfies requirement 3. Similarly, the *Candidate* being the creator of an *AnswerBook* object is its owner. However, when the answer book is submitted, the ownership is transferred to the *Checker* role, as specified by requirement 4. Assigning the *Examiner* as the owner of the *Candidate* role satisfies requirement 5(a). This also shows an example of requirement 2, when the owner of the *ExamSession* activity is not trusted with its nested *Candidate* role.

3.2 Trusted Server Selection

In the collaboration environment, each participant has highly available *trusted servers*, which provide management functionalities and object storage facilities. Each role also has a *trusted server*, which is trusted to manage entities owned by the role. A trusted server for a role is not the server where the role itself is managed, rather where the role manages the entities that it owns. There can be meta policies, which govern how a trusted server is selected for a role.

1. The default rule for selecting a role's trusted server is to select its owner's trusted server.
2. The collaboration definition can explicitly specify that

the role’s trusted server must be selected from trusted servers of its participants. Different policies can be employed to assign the trusted server. For example, the server of the first member admitted to a role, a common server from participants’ trusted servers, or a server which is trusted by the largest number of participants.

4 Consistency Issues in Decentralized Coordination

In our model, collaboration entities may be managed at different nodes. In such situations, there is no single node that has a complete view of the system state. Our specification model, on the other hand, assumes a centralized view of the system state with serialized evaluation of operation preconditions². In ensuring correctness in decentralized enforcement of coordination constraints, our goal is: *to allow only those sequence of operations that would be permitted by a centralized controller.*

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

- Events are communicated using either the *query (pull)* or *notification (push)* model. Due to delays in event notification, a role may have a stale view of the state of another entity on which its precondition depends.
- While a role’s operation precondition is being evaluated, the state of other entities, on which it depends, may change.

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

```

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

```

With a centralized coordinator, the evaluation of the conditions is serialized ensuring that as soon as either of the operation is started, the precondition of the other operation becomes false. In case of decentralized management of roles, where *r1* and *r2* are managed on different nodes, delay in the notification of the *start* event may result in both roles evaluating their preconditions to true resulting in an inconsistent collaboration state. The problem is aggravated by the fact that preconditions can be complex with dependencies involving a large number of operations of other roles.

²Evaluation of an operation’s precondition and increment of its *start* event is atomic

Note that a role is always completely managed at one node. Therefore inter-dependencies among operations of the same role will not cause inconsistencies in precondition evaluation.

To enforce consistent precondition evaluation, we must ensure that when an operation’s precondition evaluates to true, the state of other entities on which it depends, does not changing concurrently in a way as to make it false. A simple solution to the consistency problem is to enforce mutual exclusion in precondition evaluation among all entities, allowing only a single entity to evaluate operation preconditions at a time. However, this is too restrictive as it limits any concurrency in the system. A less restrictive solution is that while a role is evaluating its precondition, we block only the evaluation of those operations’ preconditions that may potentially affect its consistency. We refer to this set of operations as the *dependency set* of that operation. This set includes all those operations whose events appear in its precondition.

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

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

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

```

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

```

The *put* operation’s precondition cannot change from true to false as the count of *put.start* cannot increase during its own precondition evaluation, and the *get.finish* count is monotonically increasing. In this case the *put* operation’s precondition is *independently consistent* with respect to the *get* operation.

Dependently Consistent: If the precondition evaluates to true, and if it can be guaranteed that an operation on which it depends can not concurrently change its state, we say the precondition is *dependently consistent* with respect to

that operation. The following example specifies two operations that must be performed in strict alternation. Therefore, when the precondition of one operation is true, that of the other is false, ensuring that its state does not change while the first operation’s precondition is being evaluated.

```

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

```

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

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

```

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

```

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

We use a model checking tool, SPIN, in conjunction with the high level specification expressed in XML to determine the minimal dependency sets for role operations. The dependency set is further pruned dynamically when a predicate in an operation precondition becomes *stable* at runtime. A predicate is stable when its value can not change any further. For example, once the predicate $\#(op.start) < 5$ becomes false, it cannot become true as event counts are monotonically increasing.

Once the *minimal dependency set* for a role’s operation is determined, we need to ensure mutual exclusion in precondition evaluation. We find an entity that can be trusted by that role and all roles whose operations appear in the minimal dependency set and assign it the responsibility of arbitrating mutual exclusion. In our collaboration model, dependencies among role operations are confined within an activity ensuring that an operation precondition does not depend on an operation defined in other activities. Moreover, the owner of an activity has a vested interest in ensuring coordination consistency within the activity. The activity owner is therefore designated as the arbitrator for mutual exclusion in the evaluation of its roles’ operation preconditions.

5 Trust Issues in Event Based Coordination

Correct enforcement of security policies depends on the integrity of the events exchanged among the distributed

entities. The middleware derives two event related policies from the collaboration specification for all collaboration entities. *Event subscription policy* specifies the valid subscribers of an event generated by an entity. *Event notification policy* specifies the valid notifier for each subscribed event. Authenticating the notifier cannot guarantee that the notifier is not *falsifying* or *omitting* events. That is, the notifier may not be trusted for a received event or may not send an event when it may be to its advantage. In this section, we discuss trust issues related only to events generated by role operations. In our trust model, a role is always trusted for its membership information. In our middleware, event communication is based on Java RMI, which ensures synchronous delivery of events.

Interaction Model: As all tasks within our collaboration model are specified as role operations, a user may not need to know which objects are invoked as part of a role operation. To maintain the consistency of roles’ state in the execution environment, we impose the restriction that all the interactions among users and objects must be initiated through a role. A role operation can result in a method invocation, or initiation of a “user-object session” through which a user can have extended interactions with the object. Figure 4 represents the interaction model in our collaboration systems, where a user invokes a role operation (arrow 1), which may invoke an object (indicated by dashed arrows 2, 3). Completion, failure, or initiation of a “user-object session” for this operation is notified to the user (arrow 4). If a “user-object session” is initiated, user can have multiple direct interactions with the object (arrow 5).³

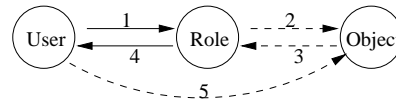


Figure 4. Interaction model

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

³In this paper, for simplicity of discussion related to trust issues, we generalize that each operation only interacts with a single object. This generalization can be extended for multiple object interactions, as an adapter object can be constructed to handle issues related to multiple objects’ method invocations.

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

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

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

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

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

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

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

Ensuring Trust in Event Communication: Here, we present solutions adopted by our distributed middleware for the above three cases:

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

The problem in Example 2 occurs as role *r3* wrongfully trusts role *r2* for enforcing its operation *op2*'s precondition.

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

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

6 Construction of Collaboration Environments

Figure 5 illustrates the steps in building the runtime collaboration environment using our middleware. Tools are provided to collaboration designers for developing collaboration specifications. These tools perform various consistency checks on the specification: such as unreachable operations. Ownership assignments are checked for any conflicts with specified security constraints such as confidentiality, access control, and role related policies. Additionally, coordination dependencies are analyzed to determine the minimal dependency sets for role operations, and it is checked whether a trusted entity can be found to arbitrate mutual exclusion in evaluating operation preconditions.

In the next step, various policy templates are derived from the specification for object access control, event subscriptions and notifications, role management, and meta policies like ownership management. The middleware provides generic managers for roles, activities, and objects. Protocols for secure interactions among users and these managers are supported by the middleware. When an activity is instantiated, policy modules are created from the corresponding policy templates. Managers for the activity and its nested entities are constructed by integrating these policy modules with generic managers, and these managers are securely distributed to their owners' sites.

7 Related Work

In groupware environments, an access control model for shared-view based GUI interface control has been presented

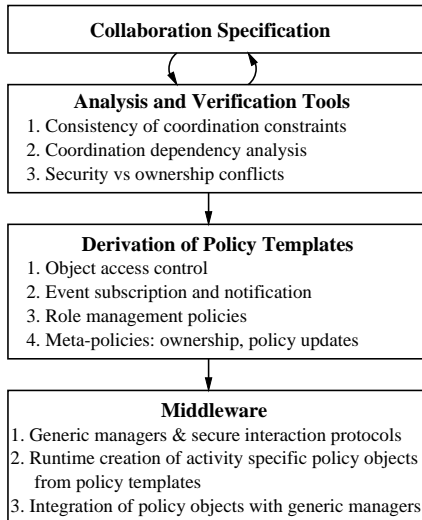


Figure 5. Steps in building the collaboration environment

in [2]. In workflow, database oriented security is prevalent. These systems, including COCA [4] and DCWPL [1], do not address secure management of the runtime systems. Security and privacy risks in distributed systems caused by protocol dependency in partial order time is addressed in [8], where to ensure event causality, a hardware based solution using “Sealed Vector Timestamp” is provided. In [6], several algorithms to prevent event forgery and denial of events by manipulation of event causality have been presented. In contrast, we identify nodes that can be trusted for various management functions and event generation. A distributed and decentralized management of role memberships in a distributed service model is presented in [3]. In comparison, we construct collaboration systems from specifications and assign management tasks to trusted entities at runtime based on a trust model. In [5], a suit of mechanisms are provided to support security policies related to group session security requirements. In contrast, we address the requirements of coordination, access control, confidentiality, and secure management of collaboration systems.

8 Conclusion

The focus of our work is on automated realization of distributed collaboration systems from their high level specification of security and coordination policies. In this paper, we have addressed several security issues that arise in decentralized management of collaboration systems and have presented the solutions that we have implemented in our policy driven middleware. We have developed a trust model to select appropriate nodes to enforce various policies con-

cerning access control of shared objects, user assignment to roles, activity creation, and other management related tasks. Our earlier specification model [11] has been extended to support confidentiality requirements and fine grained ownership assignment at entity level. As dynamic security policies are tied to coordination constraints, we have presented a mechanism for ensuring correctness in decentralized enforcement of such constraints. Our middleware prevents falsification and omission of events by identifying events whose sources cannot be trusted and subscribing corroborating events in such cases. Tools and middleware functionalities required for the construction of collaboration systems from their high level specification have been presented.

References

- [1] M. Corts and P. Mishra. DCWPL: a programming language for describing collaborative work. In *Proc. of CSCW'96*, pages 21 – 29, November 1996.
- [2] P. Dewan and H. Shen. Controlling access in multiuser interfaces. *ACM Transaction Computer-Human Interaction*, 5(1):34 – 62, March 1998.
- [3] R. Hayton, J. Bacon, and K. Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, pages 3 – 14, 1998.
- [4] D. Li and R. Muntz. COCA: Collaborative Objects Coordination Architecture. In *Proc. of CSCW'98*, pages 179–188, 1998.
- [5] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proc. of the 8th USENIX Security Symposium*, pages 99–114., August 1999.
- [6] M. Reiter and L. Gong. Securing Causal Relationships in Distributed Systems. *The Computer Journal*, 38(8):633–642, 1995.
- [7] P. Roberts and J.-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proc. of IFIP Congress*, pages 981–986, 1977.
- [8] S. W. Smith and J. Tygar. Security and privacy for partial order time. In *Proc. of Parallel and Distributed Computing Systems Conference*, pages 70 – 77, October 1994.
- [9] R. K. Thomas and R. S. Sandhu. Conceptual Foundations for a Model of Task-based Authorizations. In *In Proceedings of IEEE Computer Security Foundations Workshop*, pages 66–79, 1994.
- [10] A. Tripathi, T. Ahmed, and R. Kumar. Specification and Implementation of Secure Distributed Collaboration Systems. Technical report, Dept. of Computer Science, Univ. of Minnesota, Sept. 2001. TR 01-039, Available at <http://www.cs.umn.edu/Ajanta>.
- [11] A. Tripathi, T. Ahmed, R. Kumar, and S. Jaman. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proc. of ICDCS'2002*, pages 393 – 400, July 2002.
- [12] M. E. Zurko and R. T. Simon. User-centered security. In *Proceedings of the UCLA conference on New security paradigms workshops*, pages 27 – 33, 1996.