

Toward Minimal Number of Participants for Finite State Verification of a Role Based Model *

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

ABSTRACT

We have developed a role based specification model for distributed collaboration systems for coordination and security policies. The specification of a collaboration system is converted to finite state based verification models and are verified using a model checker for correctness and consistency. In this paper, we present a framework to find a minimal number of participants for the model checker. The minimal number of participants derived by the framework is sufficient to verify a given set of requirements. Importantly, we need to ensure that the requirements are not violated if more than the minimal number of users participate in the collaboration.

1. INTRODUCTION

The goal of our research is to generate a specification for a distributed collaboration system based on a given set of requirements. Based on these pre-generated specifications collaboration systems are realized using a policy driven middleware [13]. Existing research in specification of collaboration systems are primarily concerned with requirements related to interaction or coordination constraints [8, 4]. However, security requirements are also important concern in distributed collaboration systems. We have developed a role based specification model for security and coordination policies in decentralized management of collaboration systems. A challenge of our research is to verify that a specification can satisfy the security and coordination policies.

We have utilized an existing model checking tool SPIN [6] for finite state based verification of collaboration specification [1]. We treat collaboration specifications as executable. A finite state based verification model that corresponds to a specification resembles a reactive system requiring users interaction. This one to one correspondence between a specification and a verification model results in easy translation

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

of the specification to the verification model. As the verification model resembles the runtime system, the interacting participants' identities need to be modeled as part of the verification model. Due to search space problem of finite state based verification techniques, we need a bound on the number of participants for a verification model. However, for a specification, all roles may not have maximum membership cardinality constraints. Any number of activity instances may be created, which results in an unbounded number of roles. Additionally, there may be no bound on the number of times a role member can access objects or create objects thus creating an unbounded number of channels through which information can leak. Compared to other research in finite state based model checking, such as verification of workflow processes [5, 7], program analysis [3] and protocol verification [10], our verification of role based security specifications requires a bound on the number of participants for a verification. A similar challenge of ensuring completeness in verifying security protocols using model checking is discussed in [9].

In this paper, our goal is to find a minimal number of users per role whose unique identities are to be modeled to verify a given set of requirements for a specification. A minimal number of participants implies that if a requirement is not violated with the minimal number of participants, it will not be violated with a larger number of participants. There are two distinct inter-related problems that we need to address to find a minimal number of participants for verification. First, we need to devise an algorithm to find a minimal number of users for a set of requirements. Second, we need to prove that the number derived from the algorithm is correct, that is for a larger number of users the verified requirements will not be violated.

The next section presents the overview of our role based specification model. In Section 3, we discuss our approach of using the model checker SPIN. Section 4 presents the background on our framework for finding the minimal number of participants. Section 5 discusses case studies for the framework to find a minimal number of participants. The framework and its completeness proof are discussed in Section 6.

2. OVERVIEW: ROLE BASED SPECIFICATION MODEL

In our collaboration model, an activity defines how a group of users cooperate toward some common objectives by con-

ducting their individual tasks on a set of shared objects. In an activity, users are represented by their roles, and roles are assigned privileges to perform certain tasks.

An *activity* is an abstraction of a collaboration session, which provides a scope for objects, roles, and privileges in the collaboration. 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.

Based on our collaboration model, we have devised an XML schema, which facilitates collaboration specifications. In this section we present definition of various constructs in our role based specification model, which is followed by an example specification of a collaborative course activity. However, rather than using XML, we use a notation that is simple to read and conceptually easy to follow. In Figure 1, the syntax for the XML schema is shown in EBNF like format, where [] represents optional terms.

An *activity template* specifies a generic collaboration pattern among a set of roles using some shared objects. Any number of instances of a template can be dynamically and concurrently created.

2.1 Example Specification

A course activity contains three roles: *Instructor*, *Assistant*, and *Student*. These roles have disjoint members and can access, i.e., read/write a *BulletinBoard*. A course activity also has a nested *Examination* activity template with four roles: *Grader*, *Examiner*, *Examinee*, and *Approver*. Members of the *Instructor* and the *Assistant* roles are admitted or assigned to the *Grader* role of an instance of the *Examination* activity, when the activity instance is created. Similarly, all members of the *Student* role of a course are admitted to the *Examinee* role in any nested examination activities. On the other hand, a member of the *Instructor* role can join the *Examiner* role after the *Examination* activity instance is created.

Within an *Examination* activity, 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 the *Candidate* role. A reference to the *ExamPaper* object is passed as a parameter 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 in each exam session.

Task modeling is an integral part of collaboration specification in our role model. In our specification model, coordination policies including task synchronization and task-flow constraints are specified with roles. Such specifications are unable to provide the global view of task and activity dependencies to a collaboration designer. The designer may like to verify task-flow requirements independent of other role constraints, with an alternative form of expression. To facilitate such checks during the design, task-flow requirements can be expressed using *path expression* [2] constructs, such as **sequence** (;) and **selection** (()) with a **count**

(:n) restrictor, where n can be a constant, or “+” representing one or more and “*” representing unbounded.

The task-flow requirement for the *Examination* activity is expressed as below which requires that a *SetPaper* operation is performed before the *ApprovePaper* operation, an *ExamSession* activity can be started only after an *ApprovePaper*, and the number of exam-sessions has to be equal to the cardinality of the *Examinee* role before the *Examination* terminates. Second path, for the *ExamSession*, specifies that the *Write* can be performed one or more times 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.

```
Course := Instructor.StartExam;
Examination := Examiner.SetPaper; Approver.ApprovePaper;
                Examinee.ExamSession:#member(Examinee)
ExamSession := Candidate.OpenExam; Candidate.Write:+;
                Candidate.Submit; Checker.Grade
```

The designed need to take into account role related constraints as part of security policies. These policies may be specified as confidentiality or resource access policies. are several role constraints specified for the example in Figure 2, we choose the following two role related requirements to be verified.

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

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

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

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

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

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

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

AL6 *A participant of the examinee role can only modify his/her answer book before end of his/her exam-session.*

2.2 Specification Model

A skeleton specification of the *Course* activity template, with a complete specification of the nested *Examination* activity is shown in Figure 2. In the specification model, an

```

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

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

OperationDef → Operation opId [PreCondition Condition] [Action actionDef]

ReactionDef → Reaction opId PreCondition Condition [Action ActionDef]

ActionDef → NewObjectDef [ObjectInvocationDef] [NewActivityDef]
           { ChangeOwner objId RoleId }

NewObjectDef → objId = new Object codebase

NewActivityDef → activityId = new Activity templateId
                { PassedObject ObjContextId } { MemberAssignment roleId }

ObjectInvocationDef → MethodInvocation objId methodSignature [retObjId]
                    | ObjectInterface codebase objId { methodSignature }

Condition → RoleCondition | OperationCondition | Condition BinOp Condition
           | ! Condition

RoleCondition → #RoleMemberList Relation Count | member( UserID, roleId)

OperationCondition → #opId Relation Count | ComplexEventCondition

ComplexEventCondition → Exp Relation Count

RoleMemberList → RoleTerm { SetOp RoleTerm }
RoleTerm → ( RoleMemberList ) | member(roleId)
BinOp → ^ | v
Relation → > | < | = | <= | >= | ≠
Exp → Term { AddOp Term }
AddOp → + | -
Term → Term { MulOp Factor } | Factor
MulOp → × | ÷
Factor → ( Exp ) | #opId | Count
SetOp → ∩ | ∪ | \
Count → 0 | 1 | ... | N

```

Figure 1: Pseudo syntax of the specification language

entity (such as activity, object, or role) encapsulated in the scope of an activity can be referenced by a fully qualified name. Within an activity, one can refer to its current instance using the pseudo variable *thisActivity* and its parent activity using *parentActivity*. The user executing an operation is identified by *thisUser*. Within a role's operation, one can refer to the role by *thisRole*.

Events and event counters are used in our model for specifying coordination and dynamic security policies. Conditions specified for activity termination, role admission constraints and operation preconditions can be based on event count based predicates. Event types are related to different kinds of entities such as activities, roles, and operations. For example, instantiation and termination of an activity, execution of a role operation and admission of users in a given role represent different types of events.

Events are implicitly generated by the system, based on the

model presented in [11]. Related to each activity and each role operation are two types of events: *start* and *finish*. A *count* operator, #, returns the number of occurrence of a given event type.

One can also specify a derived event type by filtering an event list based on predicates on event attributes. For example, for a role operation execution, using the expression `opName.start(invoker=thisUser)`, we can define a filter based on the operation invoker identity. The expression `#(opName.start(invoker=thisUser))` counts the number of times the invoker has invoked this operation.

A role defines a set of tasks, and these role specific tasks are termed *operations*. A role specification includes role name, *reflected* roles if any, role admission, validation, and activation constraints, and role operations with their preconditions. These constraints and preconditions facilitate dynamic access control policies. In the specification, a boolean

operations defined within the same activity scope.

2.2.1 Role Admission Constraints

Role admission precondition must be true when a user is admitted to a role. For example, the *Assistant* role in Figure ?? can have several admission constraints:

A role cardinality constraint requires that the member count for this role cannot exceed one; A *static separation of duty* constraint requiring that the assistant can not be a student in the course; and A user is admitted to this role only when at least one member is present in the *Instructor* role.

2.2.2 Role Validation Constraints

A validation constraint differs from an admission constraint as follow: (1) It can only be specified based on role membership related predicates; (2) It needs to be satisfied not only when a user is admitted to a role but also when a member initiates any role operations; and (3) It is also verified when any membership related query is invoked on the role. In

2.2.3 Role Admission

Our specification model supports two mechanisms to admit members in a parent activity role to a child activity role. As described earlier the first mechanism, *static role assignment* or *role reflection*, is specified using the *Reflect* construct. In role reflection, all members of specified roles in the parent activity become members of a role in the child activity when that activity is created. In Figure 2, using role reflection, all members of the *Student* role in the parent activity are admitted to the *Examinee* role, subject to the *Examinee* role's admission constraints. 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 users for role admission are specified at the time of activity instantiation. The template definition uses the *Assign-dRoles* tag to specify which roles must be assigned at the time of activity instantiation, as specified for the *Candidate* role of the *ExamSession* activity in Figure 2. When an examinee invokes this *StartExam* operation, an instance of the *ExamSession* template is created and the participant creating the instance is dynamically assigned to the *Candidate* role.

2.2.4 Role Operation Specification

An operation specification includes a name, and may include a precondition and an action. The precondition must be true when the operation is invoked. The preconditions associated with operations allow one to specify coordination constraints as well as various dynamic security requirements, such as condition-based access control, dynamic “separation of duties”, context-based access control. Other than the specified operations in a role, there are four predefined operations for each role: *Join*, *Leave*, *Admit*, and *Remove*.

The action part of an operation may invoke object methods, or create a new object or a nested activity. If the action part is empty, then the operation is used primarily for coordination purposes using its *start* and *finish* events. A keyword

new is reserved for specifying creation of an object or activity. 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 *exam* object of type *ExamPaper* and an invocation of the *setQuestions* method of this object.

Preconditions also enable us to specify coordination constraints, for both *inter-role* and *intra-role* coordination. In Figure 2, a student in the *Examinee* role can not execute the *StartExam* operation until the *Approver* has approved the exam paper. The precondition for this operation also illustrates an intra-role coordination policy, which allows each member in the *Examinee* role to independently start an exam session. This illustrates the *independent participation model* for the members in the *Examinee* role. In the previous example snippet, the precondition of the *SetPaper* operation in the *Examiner* role illustrates the *cooperative participation model* where only one of the role members can execute the *SetPaper* operation.

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

2.2.5 Role Activation Constraints

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

A *dynamic separation of duty* constraint, such as a user should not activate two roles at the same time, needs to be specified as part of role activation or validation constraints. Similarly, *previous qualifications* that must be ensured during role operation invocation need to be specified as activation or validation constraints. As opposed to validation constraints, a membership related predicate in activation constraints does not validate any other membership related predicates that it depends on.

2.2.6 Role Admission

Our specification model supports two mechanisms to admit members in a parent activity role to a child activity role. As described earlier the first mechanism, *static role assignment* or *role reflection*, is specified using the *Reflect* construct. In Figure 2, using role reflection, all members of the *Student* role in the parent activity are admitted to the *Examinee* role, subject to the *Examinee* role's admission constraints.

The second mechanism is *dynamic role assignment*, in which users for role admission are specified at the time of activity instantiation. The template definition uses the *Assign-dRoles* tag to specify which roles must be assigned at the time of activity instantiation, as specified for the *Candidate* role of the *ExamSession* activity in Figure 2. When an examinee invokes this *StartExam* operation, an instance of the

ExamSession template is created and the participant creating the instance is dynamically assigned to the *Candidate* role. As specified in Figure 2, an *ExamSession* activity terminates when the *Checker* performs the *Grade* operation.

2.2.7 Termination Condition

As specified in Figure 2, an *ExamSession* activity terminates when the *Checker* performs the *Grade* operation. The *StartExam.finish* event is generated as soon as the activity *act* is instantiated; however, the *act.finish* event is generated when the *act* activity is terminated.

2.3 Meta Policy Specification

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

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

1. The template specification may indicate which role would be the owner of an entity. Meta roles, such as *Creator*, can be specified as an owner.
2. If not explicitly specified:
 - for an activity, the owner of the parent activity is the owner;
 - for a role, owner of the activity in which the role is defined becomes its default owner; and
 - the default owner of an object is the role that creates it.

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

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

3. MODEL CHECKING WITH SPIN

SPIN [6] is a model checker based on an automata theoretic approach [14]. In SPIN, a model of a system to be verified is specified in PROMELA (a Process Meta Language). Given the model of a system and a desired property of the system, SPIN converts the the model of a system and the negation of a desired system property to finite Buchi automata, which accepts infinite words thus representing non-deterministic behavior of reactive systems [14]. Next, SPIN generates a language intersection of these two automata and uses efficient depth-first search algorithm with a partial order reduction method to find a trace of the counter-example for the desired property. The desired system property can be expressed in LTL (Linear Temporal Logic) using temporal

operators **always** (\square), **eventually** (\diamond), and **until** (U). In our approach, the collaboration specification in XML is converted to PROMELA. However, additional components are needed to be added to the model to verify properties related to information flow and owner assignments.

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, in our verification model for role constraints, to verify users' admission to roles, modeling of role operations that cannot affect users movement among roles is not required. Properties related to such role operations is viewed as independent aspects and verified separately. Also, the model is composed incrementally, adding a new property to the model, given a set of properties that have been verified.

A second problem is the inter-weaving aspects of the verification requirements. Among the aspects of the properties presented in the previous section, reachability of operations, task-flow, and role based constraints are correctness properties. On the other hand, confidentiality and access leakage are security properties. The verification methodology needs to ensure that it follows a precedence among the properties it checks. Consequently, we have developed, based on the aspects of global requirements to be checked, four models for efficient verification. The models are termed *Task Model*, *Role Model*, *Information Flow Model*, *Owner Assignment Model*. These models, respectively, verify correctness of coordination constraints, role constraints, confidentiality requirements with and without assignment of administrative roles, and integrity requirements when owners are assigned. In [1], we show how the inter-weaved verification concerns are managed based on these verification models.

4. BACKGROUND TO FIND A MINIMAL NUMBER OF PARTICIPANTS

Our goal is to find a minimal number, N , of users whose identities are adequate for verification of all the requirements. If a requirement is not violated for the model with N users, it will not be violated with more than N users. If we only need to find the minimal number of users to verify a specific requirement, there may be a smaller number n' ($n' < N$) of users that is adequate to verify that requirement. Our goal is to find a bound, i.e. a minimal number, not a minimum number, of participants for verification. For a given specification, we are always concerned with a set of requirements. If there exist a minimal number for each of the requirements, then there exist a minimal number for the set of requirements.

Verification of all the requirements does not guarantee that the specification is correct and consistent. By consistent specification, we mean that all the operations can be executed, i.e., all the operations are reachable with the assigned users. An activity, in our model, can be non-terminating, but there must exist an execution path for each operation. Otherwise the operation specification is redundant. To ensure correct and consistent specification, we define the term *constraint satisfaction*.

Definition 1: *Constraint satisfaction* for an activity means

that (1) all the role operations can be executed and (2) the specified constraints can fulfill all coordination and task flow constraints as well as security requirements.

The framework to find a minimal number of participants is facilitated by several restrictions of our specification model. These restrictions are presented as facts in the following discussions..

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

Based on Fact 1, our goal is to find a minimal number of participants for the roles that do not have prerequisite membership constraints. We term these roles as *initial assignment roles*.

Definition 2: We define *equivalent classes* that partition the set of initial assignment roles R and the set of a minimal number of users U . That is *equivalent classes* $C = \{C_1, \dots, C_n\}$, where $C_i = \{R_i, |U_i|\}$, $|U_i|$ is the cardinality of the set U_i , $R_i \in R$, $U_i \in U$ and $R_i \cap R_j = \text{null}$ and $U_i \cap U_j = \text{null}$ for $i \neq j$, i.e, two equivalent classes do not share a role or a user. Our goal is to find a $|U| = U_1 + \dots + U_n = N$, such that all the requirements can be satisfied.

Goal: Given an activity template T with a set of initial assignment roles R , for each role $r \in R$, find the pair $C_i = \{R_i, N_i\}$, where $R_i \in R$, $R_i \cap R_j = \text{null}$ for $i \neq j$, such that the verification run does not miss a possible execution trial which may violate a property. That is the search space contains execution trials, which are equivalent to all possible execution trials of policy violations. Our goal is to find the equivalent classes $C = \{C_1, \dots, C_n\}$, where $C_i = \{R_i, N_i\}$ and $N_1 + \dots + N_n = N$.

Fact 2: *Parts of the specification model that require an invoker identity are specified as: (1) `member(thisUser, roleId)`, and (2) `opName(invoker = thisUser)`. These constructs are located at role admission and validation constrains and operation precondition.*

To find a minimal number, we rely on Fact 2 that none of the requirements is specified based on participants' identities but rather on participants' role memberships and their previously performed operations. We define the term *invoker credential* representing the conditions that an invoker has to satisfy to invoke a specific operation.

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

An operation can have multiple *invoker credentials* as the invoker can acquire a role membership based on memberships to different set of roles. Moreover, an operation precondition can have disjunctive conditions resulting in more than one

invoker-credentials for the operation. The idea behind the framework is that if two users u_1 and u_2 with same *invoker-credentials* can satisfy a property, a verification run with either one of them is sufficient. The goal of our framework is to find a minimal number of participants that can generate all possible *invoker-credentials* when assigned to roles in non-deterministic steps.

5. CASE STUDIES

Only operation can violate security requirements. Hence *invoker credentials* are concerned with role constrains and the operation preconditions that need to be satisfied for a role operation. In this section, based on different case studies, we discuss the *invoker credentials* and a minimal number of users assignment to achieve the *invoker credentials*. We derive minimal numbers based on the aspects of the verification models presented in [1].

5.1 Role Model

In role model, we are not concerned with operation preconditions and the *invoker credentials* are considered for roles instead of a specific operation of a role. In role model, a role R_1 has an *invoker credential* $\{R_2\}$ implies that a user has to be a member of R_2 to be a member of R_1 . Following test cases specify role based constraints for two or more roles and discuss the user assignments that is sufficient to generate equivalent classes for member assignments. To find the equivalent classes C , we need to ensure that the user assignment can generate all possible *invoker credential*.

5.1.1 Admission Constraints

In this section, we present cases with admission constraints that are specified based on prerequisite roles. The invoking user has to satisfy membership to the prerequisite roles to be a member of the role with admission constraints.

Case R1: There are only two roles R_1 and R_2 in an activity template without any admission constraints.

Without any admission constraints, these roles are initial assignment roles. The possible *invoker credentials* for R_1 are $\{\text{null}\}$ and $\{R_2\}$. To be able to generate all these *invoker credentials*, we can have an equivalent class $C_1 = \{\{R_1, R_2\}, 1\}$. In this assignment, two roles have been assigned only one member. As the member is assigned in non-deterministic step all the *invoker credentials* are satisfied.

Case R2

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

In this example, possible *invoker credentials* for R_2 is $\{R_1\}$ and for R_1 is $\{\text{null}\}$ and $\{R_2\}$. The partition of the equivalence class of roles and users cannot express the dependency imposed on users' admission as specified by this example. Due to non-deterministic step of user assignment, to satisfy all the *invoker credentials*, $C = \{\{R_1, R_2\}, 1\}$ is sufficient. For R_2 , this covers the execution trial of an *invoker credential*, $\{R_1\}$, that is not valid, i.e., a member of $\{R_1\}$ cannot join $\{R_2\}$. We can accept this C as a solution considering at

runtime the admission constraints are imposed in the verification model. This runtime check ensures that a member of R1 cannot join R2.

Primary reason behind equivalent class, which derives a minimal number for user to be assigned per role, is to reduce the state space in the verification model. In the case with admission constraints, partitioning users for a subset of the roles reduces the search space by avoiding the check for admission constraints that cannot be valid. However, the case presented here shows that all such check cannot be avoided by partitioning the users and roles.

Case R3

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

In this case, R2 has a role admission constraint. Possible *invoker credentials* for R2 is {R1} and for R1 are {null} and {R2}. To satisfy all the *invoker credentials*, $C = \{\{R1, R2\}, 1\}$ is sufficient. As users are added in non-deterministic step, all possible *invoker credentials* are created.

Case R4

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

In this case, possible *invoker credentials* for R2 is {!R1} and for R1 is {!R2}. To satisfy all the *invoker credentials*, $C1 = \{\{R1\}, 1\}$, $C2 = \{\{R2\}, 1\}$ is sufficient.

Case R5

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

This example is an inconsistent specification. In the verification model, any user assignments will report unreachable states.

Case R6

```
Role R1 { }
Role R2 { AdmissionConstraints !member(thisUser, R1) }
Role R3 { }
```

In this case, possible *invoker credentials* for R1 are {null}, {R2}, {R3}, and {R2, R3}, for R2 are {!R1, R3} and {!R1}, and for R3 are {R1}, {R2}, and {R1, R2}. For R3, {R2, R1} is a valid *invoker credential* as a user may join R1 followed by R2. To satisfy all the *invoker credentials*, $C1 = \{\{R1, R2, R3\}, 1\}$ is sufficient.

Case R7

```
Role R1 { }
Role R2 { AdmissionConstraints !member(thisUser, R1) }
Role R3 { AdmissionConstraints !member(thisUser, R2) }
```

In this case, possible *invoker credentials* for R1 are {null}, {R2}, {R3}, and {R2, R3}, and for R2 are {!R1}, {!R1, R3}, and for R3, {!R2, R1}. An equivalence class to satisfy all the *invoker credentials* is $C1 = \{\{R1, R2, R3\}, 1\}$.

Case R8

```
Role R1 { }
Role R2 { AdmissionConstraints !member(thisUser, R1) }
Role R3 { AdmissionConstraints !member(thisUser, R1) }
```

In this case, possible *invoker credentials* for R1 are {null}, {R2}, and {R2, R3}, and for R2 are {!R1}, {!R1, R3}, and for R3, {!R1} and {!R1, R2}. An equivalence class to satisfy all the *invoker credentials* is $C1 = \{\{R1, R2, R3\}, 1\}$

5.1.2 Admission Constraints: Member Count

The following cases are based on role cardinality related admission constraints.

Case R9

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

For $N > 0$, the above is an inconsistent constraint. Specification of minimum cardinality on role admission constraints is illegal.

Case R10 On the other hand, for $N > 0$, the following example shows the effect of role cardinality on the *invoker credential* credential count.

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

A role's own member count in an admission constraints represent maximum cardinality. When maximum cardinality is specified, the count on this role's equivalence class has to be incremented. In this case, $C = \{\{R1\}, N\}$, which ensures that the verification with the requirement of maximum cardinality is satisfied.

Case R11

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

In this example, admission to the role R1 depends on the member count of the role R2. Here, R2 has to have at least N member, i.e., $C = \{\{R2\}, N\}$ to ensure that R1 can have a member. Here, the *invoker credential* for R1 is $\{\#\text{member}(R2) \geq N\}$.

Case R12

```
Role R1 {
  AdmissionConstraints #( member(R2) ∩ member(R3) ) >= N }
```

In this case, R1 and R2 have to have at least N common members. If R2 and R3 are in the same partition, then

the member count for the equivalence class needs to be at least N . If they are in different partitions, which signifies that $R2$ and $R3$ cannot have common members, makes this constraint inconsistent.

Case R13

```
Role R1 {
  AdmissionConstraints #( member(R2) ∪ member(R3) ) >= N }
```

The total count for both of the partitions for $R2$ and $R3$ has to be equal to N .

5.1.3 Validation Constraints

In the following, we discuss the above cases with validation constraints. If the admission constraints are replaced and validation constraints are specified with the same constraints, for Case R1, Case R3, Case R5, the possible *invoker credentials* and the equivalence classes remains same. In the following, the effect of using validation constraints for the rest of the cases are discussed.

Case R14 This case is similar to Case R2, except role admission constraints is changed to validation constraints.

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

In this example, the possible *invoker credentials* for $R2$ is $\{R1\}$ and for $R1$ is $\{\text{null}\}$ and $\{R2\}$. If a member of $R2$ join $R1$, his/her membership to $R2$ is revoked. To satisfy all the *invoker credentials*, a solution is $C1 = \{\{R1, R2\}, 1\}$.

Case R15

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

In this example, the possible *invoker credentials* for $R2$ is $\{R1\}$ and for $R1$ is $\{R2\}$. To satisfy all the *invoker credentials*, equivalence classes $C1 = \{\{R1\}, 1\}$ and $C2 = \{\{R2\}, 1\}$ are sufficient.

Case R16

```
Role R1 { }
Role R2 { ValidationConstraints !member(thisUser, R1) }
Role R3 { }
```

In this example, possible *invoker credentials* for $R1$ are $\{\text{null}\}$, $\{R2\}$, $\{R3\}$ and $\{R2, R3\}$, for $R2$ are $\{R1, R3\}$ and $\{R1\}$, and for $R3$ are $\{R1\}$, and $\{R2\}$. In comparison with the case when the constraints for $R2$ are specified as admission constraints, $\{R2, R1\}$ is not a valid *invoker credential* with validation constraints. In this case, if a user joins $R1$ followed by $R2$, it membership to $R2$ is revoked. However, being a member of $R2$, a user's membership to $R1$ is not refused, To satisfy all the *invoker credentials*, an equivalence class is $C1 = \{\{R1, R2, R3\}, 1\}$.

Case R17

```
Role R1 { }
Role R2 { ValidationConstraints !member(thisUser, R1) }
Role R3 { ValidationConstraints !member(thisUser, R2) }
```

In this example case, possible *invoker credentials* are, for $R1$, $\{\text{null}\}$, $\{R2\}$, $\{R3\}$, $\{R2, R3\}$, for $R2$, $\{R1\}$, $\{R1, R3\}$, and for $R3$, $\{R2\}$ and $\{R2, R1\}$. To satisfy all the *invoker credentials*, an equivalence class is $C1 = \{\{R1, R2, R3\}, 1\}$.

Case R18

```
Role R1 { }
Role R2 { ValidationConstraints !member(thisUser, R1) }
Role R3 { ValidationConstraints !member(thisUser, R1) }
```

In this example, possible *invoker credentials* for $R1$ are $\{\text{null}\}$, $\{R2\}$, and $\{R2, R3\}$, for $R2$ are $\{R1\}$, $\{R1, R3\}$, and for $R3$ are $\{R1\}$ and $\{R1, R2\}$. To satisfy all the *invoker credentials*, an equivalence class is $C1 = \{\{R1, R2, R3\}, 1\}$ as the user is added in non-deterministic step.

Case R19

```
Role R1 { }
Role R2 { ValidationConstraints !member(thisUser, R1) }
Role R3 { ValidationConstraints member(thisUser, R2) }
```

In this case, the possible *invoker credentials* for $R1$ are $\{\text{null}\}$, $\{R2\}$, $\{R3\}$, $\{R2, R3\}$, for $R2$ are $\{R1\}$, $\{R1, R3\}$, and for $R3$ are $\{R2, R1\}$. To satisfy all the *invoker credentials*, an equivalence class is $C1 = \{\{R1, R2, R3\}, 1\}$.

5.2 Access Leakage Model

Access leakage related requirements are violated when a user with some credentials can access a restricted object. In this section, we are only concerned with owners who cannot violate role constraints. Access leakage can result from incorrect specification in two distinct cases: (1) the user is able to join a role with the access rights that the designer has not intended to, or (2) the access right is specified using a dynamic policy that depends on operation related events, and the access constraints are satisfied in unforeseen situations. To verify access leakage related requirements, operation preconditions as well as role activation, admission, and validation constraints are modeled in our verification model.

In the following discussions, role activation constraints are not discussed separately as the possible *invokers credentials* due to these constraints remains same with operation preconditions. For admission and validation constraints, *invoker credentials* remains same as discussed previously as part of the role model. However, in this mode, as the *invoker credentials* are derived for a specific operation, the *invoker credentials* for the operation require additional conditions related to the operation's preconditions.

5.2.1 Precondition

For operation preconditions, we are concerned with intra-role coordination constraints. These constraints are specified using the attribute `thisUser` in the operation related predicates of an operation precondition. A motivation in

this section is to find a minimal number that satisfies all the intra-role coordination requirements.

Fact 3: An operation action can be (1) Empty, (2) Invocation of methods on an object, (3) Creation of an object, (4) Creation of an activity. Only certain combinations of the above actions are valid for an operation.

Fact 4: There can be two types of event count based predicates that are specified using `thisUser` attribute. These are:

1. `#(event) Relation Number;`
E.g. `#op(invoker=thisUser)=0`, and
2. `(Expression on #Event) Relation Number;`
E.g., `#op1(invoker=thisUser) - #op2=0`; and
 `#op1(invoker=thisUser) - #op2(invoker=thisUser)=0`;

The minimal number of users that need to perform an operation depends on the effects of the `thisUser` attribute in preconditions. These effects are: (1) the user's identity specified with `thisUser` attribute is passed to a new activity or (2) other operations depend on who performed this operation. These effects result from policies related to coordinating multiple participants within a role or policies related to ordering of a specific user's actions. For example, a policy can be that users who have performed specific operations cannot perform other operations within a role. For verification to find counterexamples that can violate these types of policies, more than one user is required to enable the operations.

Case A1: The following precondition specifies that an operation can be performed only N times. In the following cases, the operation has no action.

```
Operation Op1 {
  Precondition #Op1(invoker=thisUser) <= N }
```

With $N=0$, this implies this operation can be performed only once. If no other operation depends on this operation, then the count for the equivalence class for this role has to be at-least 1.

Case A2: This example is similar to Case A1. However, in the following case, the operation has an action, which is a method invocation.

```
Operation Op1 {
  Precondition #Op1(invoker=thisUser) = 0
  Action obj,method() }
```

As users are added in non-deterministic step, in the verification model all users with same *invoker credentials* can access the object. Based on only this action, the count of 1 for the equivalence class is sufficient. We will discuss the information flow related cases in a later section.

Case A3: In the following cases, other operation depends on an operation with precondition based on `thisUser` attribute.

```
Role R1 {
  Operation Op1
  Precondition #Op1(invoker=thisUser) = 0 }
Role R2 {
  Operation Op2
  Precondition #Op1 = 0
  ..... }
```

When operation op2 has no action, as the users are performing op1 with non-deterministic steps, op2 is effect free. When operation op2 invokes a method, invocation of op2 depends on who performs op1. However, we ensure that all possible *invoker credentials* can perform op1. Thus we ensure that all possible *invoker credentials* can perform op2. Similarly, when operation op2 creates an object, the equivalence classes for R1 and R2 need to ensure all possible *invoker credentials*.

Case A4:

```
Role R1 {
  Operation Op1
  Precondition #Op1(invoker=thisUser) = 0 }
Role R2 {
  Operation Op2
  Precondition #Op1(invoker=thisUser) = 0
  ..... }
```

When the operation op2 has no action, same user has to perform both the operations op1 and op2. If the same user cannot be a member of both R1 and R2, this is an inconsistent specification. In the verification mode, this situation is reported by unreachable states. When the operation op2 invokes a method, invocation of op2 depends on who performs op1. We need to ensure all possible *invoker credentials* can perform Op1. Similarly, we have to ensure that all possible *invoker credentials* can perform Op2. For other types of actions, we have same arguments for this Case A4 as well as Case A3

Case A5: An operation can depend on a user's invocation of another operation within the same role as shown by the following example.

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

The *invoker credentials* for Op2 is $\{ \#Op1(invoker=thisUser)=0 \}$. This require at-least a member to perform Op1. Similar to the cases with different roles, the assignment needs to ensure all possible *invoker credentials*. If the role has no other constraints, for this specific *invoker credentials* a count of 1 for R1 is sufficient to check the requirement that if a user performs Op1 he/she cannot perform the other operation, Op2.

Case A6 There can be preconditions that are specified to ensure that a user can perform either of two operations as shown in the following example.

```

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

```

For either of the operations, a count of 1 for R1 is sufficient to check the requirement that if a user performs an operation he/she cannot perform the other. As the operations are performed in non-deterministic steps, assignment of one member can ensure that the verification model checks both the execution scenarios where either Op1 or Op2 is performed.

Case A7 There can be preconditions specified to ensure that an operation can be performed if two different users has performed another operation.

```

Role R1 {
  Operation Op1
    Precondition #Op1(invoker=thisUser)= 0 }
Role R2 {
  Operation Op2
    Precondition #Op1= 2 }
    ..... }

```

In this case, the equivalence class of R1 has to have at least two users. Moreover, without any other constraints, a member R1 can be a member of R2. Hence $C1 = \{R1, R2\}$, 2}.

Case A8 There can be preconditions specified to ensure that an operation Op2 can be performed if another user has performed the operation Op1.

```

Role R1 {
  Operation Op1
    Precondition #Op2(invoker=thisUser)= 0
  Operation Op2
    Precondition #Op1(invoker=thisUser)= 0  $\wedge$  #Op1= 1}

```

In this case, at least two users are required to satisfy Op2's *invoker credentials* $\{\#Op1(invoker=thisUser)= 0 \wedge \#Op1= 1\}$.

Case A9 There can be preconditions specified to ensure that a operation can be performed if two other operations have not been performed.

```

Role R1 {
  Operation Op1
    Precondition #Op2(invoker=thisUser)= 0
     $\wedge$  #Op3(invoker=thisUser)= 0}
  Operation Op2
    Precondition #Op1(invoker=thisUser)= 0
     $\wedge$  #Op3(invoker=thisUser)= 0}
  Operation Op3
    Precondition #Op1(invoker=thisUser)= 0
     $\wedge$  #Op2(invoker=thisUser)= 0}

```

In this case, a single user is sufficient to satisfy all the operations *invoker credentials*.

Case A10 If the example in Case A8 is extended with another operation R4 that requires all other operations are performed, the equivalence class for R1 requires at least 3 members.

```

Role R1 {
  Operation Op1
    Precondition #Op2(invoker=thisUser)= 0
     $\wedge$  #Op3(invoker=thisUser)= 0}
  Op2
    Precondition #Op1(invoker=thisUser)= 0
     $\wedge$  #Op3(invoker=thisUser)= 0}
  Operation Op3
    Precondition #Op1(invoker=thisUser)= 0
     $\wedge$  #Op2(invoker=thisUser)= 0}
  Operation Op4
    Precondition #Op1> 0  $\wedge$  #Op2> 0  $\wedge$  #Op3> 0 }

```

Here, the count 3 is derived based on the precondition of operation Op4, which depend on three other operation Op1, Op2, and Op3. These three other operations has preconditions based on operation predicates with *thisUser* attribute.

5.3 Information Flow Model

For the information flow model, the framework also considers operation preconditions for *invoker credentials* specified using *thisUser* primitives in operation preconditions. In our model, there are no constructs or mechanisms for entities in concurrent activities to reference each other. Only way entities in concurrent activities can interact is through shared objects. In this case, we need to ensure that users with all possible *invoker credentials* can share the object. Hence, the *invoker credentials* for the information flow verification model also considers the channels that are created due to shared objects among participants. These channels enable participants of concurrent activities to share information.

Case I1 In the following case, the operation has an action, which creates an object.

```

Operation Op1 {
  Action new Object ObjectType1 }

```

If a requirement specifies that only one instance of an object can be created, $N = 1$ is sufficient. Within this example, there is no Information flow when two objects of the same type are created.

Case I2 In the following cases, the operation has an action, which creates an activity.

```

Operation Op1 {
  Action new Activity Activity1 }

```

As the instantiated activities in this example has no shared objects, there are no shared channels.

Case I3 In the following cases, the operation has an action, which creates an activity and assign users to instantiated activities.

```

Role R1 {
  Operation Op1 {
    Precondition #Op1(invoker=thisUser) = 0
    Action new Activity Activity1(R2=thisUser)}
}

```

This is an inconsistent specification as the R2 roles in instantiated activities cannot have the same members due to precondition. However, effect of assignment of users to instantiated activities in the number of *invoker credentials* is covered in role model.

Case I4 In the following cases, the operation has an action, which creates an activity and passes shared objects to nested activities.

```

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

```

In this case, a single object is shared among concurrent activities instantiated from the same activity template. We need to ensure that at-least two users with same *invoker credentials* for the role R1 exist among these concurrent activities. That is for the role R1, N is at-least 2. This will ensure that information can flow within a role. If there are two different *invoker credentials* for R1, for example a member of this role R1 needs to be either a member of R2 or R3 resulting in *invoker credentials* {R2} and {R3}, we need to ensure that members of R2 as well as members of R3 can share. This requires that R1 have at least two members from the roles R2 and R3 as members.

6. FRAMEWORK TO FIND A MINIMAL NUMBER

From the case studies of the previous section, we see that given a case we can find an equivalence class for user assignment in roles that generates all possible *invoker credentials*. In the role model, to generate all possible *invoker credentials* the generation of the equivalent classes has four distinct scenarios:

1. A single partition with assignment of a single user in all the role can ensure all possible *invoker credentials* as shown with the cases R1, R2, R3, R6, R7, R8, R14, R16, R17, R18, and R19.
2. The roles are partitioned into distinct equivalent classes due to constraints that require disjoint members as shown with the cases R4 and R15.
3. The member count of the partition has to be incremented to satisfy specific constraints as shown with the cases R10, R11, R12, and R13. In our framework for minimal count, we can mark these partitions with the required member count. At the end when a minimal number is derived, we need to ensure that the required count is satisfied.
4. Lastly, the specified constraints cannot be satisfied and some roles cannot have members due to inconsistent

specification as shown with the cases R5 and R9. Similar inconsistent specification may arise due to other constraints that control flow of member in a given role. For example, in the Case R12, if the roles R1 and R2 are in different partitions, the constraints in this example cannot be satisfied. To specify a constraint that require common members for R1 and R2, we need to check the other constraints that put R1 and R2 in different partitions. Either this or the other constraints need to be changed to have a consistent specification.

In the access leakage model, to generate all possible *invoker credentials* for a specific operation, we need to generate all possible *invoker credentials* for the operations or roles specified in the preconditions. Lastly, in the information flow model, when sharing of object may results in information channels, two users with same *invoker credentials* need to be present to ensure information flow among users with same *invoker credentials*. We also see that the participant count in the equivalence class of a role can increase as we incrementally add verification models for access leakage and information flow properties.

6.1 An Algorithm to Find a Minimal Number

It should be noted that we are only concerned with specific types of requirements presented in [12] when we are looking for a minimal number of participants. Other types of requirements, which may exist but are not presented in [12], are not in concern for the framework to find a minimal number of participants. Based on the case studies, in this section we present a simple algorithm for the framework to find a minimal number for a specification.

Followings are the steps to find a minimal number of participants for an activity template:

1. initially, all the roles in the activity template are assigned a *member-count* based on maximum cardinality constraints, starting from the roles defined in the inner most activity templates. If there is no maximum cardinality constraints for a role, a member-count of 1 is assigned to the role.
2. Starting from the roles defined in the inner most activity templates, for each operation based on its precondition, a *member-count* is calculated that can generate all possible *invoker credentials*. This *member-count* can be for the role where the operation is defined or other roles that are referred in the precondition. As seen in the case studies, this count depends on the event counts in the predicates of the precondition. If the new *member-count* is greater than existing *member count* for a role, the role's *member count* is incremented with the new count.
3. The nested activity templates that share objects from parent activity template are marked. The *member-count* of the roles in roles in the parent activity template who participate in these nested activities are incremented by 1. This ensures that there are always more than one instance of a specific invoker credentials. This step is also initiated starting from the roles

defined in the inner most activity templates. At the end of this step, a minimal number is assigned for each of the *initial assignment roles*.

4. The last step is to ensure that the *initial assignment roles* can share the same participants. Based on static separation of duties constraints, specified using role admission and validation constraints, the *initial assignment roles* are partitioned into equivalence classes. Each class contains a subset of the *initial assignment roles* and a participant count, which is the greatest *member count* of the roles in the subset.

These steps are discussed below using our example *Course* activity template in Figure 2:

1. The *Candidate* and *Checker* roles in the inner most *ExamSession* activity template have a maximum cardinality constraint of 1. As all other roles have no cardinality constraints, a member-count of 1 is assigned to each of the roles.
2. This in turn enable us to find a minimal number of participants. In our example, the *ApprovePaper* and the *StartExam* operations have invoker identity dependent “intra-role coordination” requirements. The *invoker credentials* for the *ApprovePaper* operation are:

$$\{Adm2, \#(SetPaper.finish(invoker=thisUser))=0\}$$
 and

$$\{Instructor, \#(SetPaper.finish(invoker=thisUser))=0\}.$$
 Addition of the operation precondition in *invoker credential* does not effect the number of *invoker credentials* for this operation as there is only one operation for this role. Similarly, the *StartExam* operation’s *invoker credential* count remains same with the addition of the operation preconditions.
3. Due to the shared objects – *BulletinBoard* and *ExamPaper* – concurrent *Examination* and *ExamSession* activity instances share information channels. However, it is specified that only one instance of the *Examination* activity can be created. For probable channels among multiple instances of *ExamSession* activities, at least two of the *ExamSession* instances need to be created. For the minimal count, this requires at least two members for the *Examinee* role, which in turn requires two members for the *Student* role.
4. As the roles of the top *Course* activity template have disjoint members without any cardinality constraints. The *Approver* role can have a member who is a member of the *Instructor* role or is a participant assigned by the *Adm2* role. These result in a minimal number for initial participants to be 5 where *Instructor* and *Approver* can have same participants. Hence the final minimal count based on our framework is 5 with 2 is *Student*, 1 in *Assistant*, and shared 2 in *Instructor* and *Approver* roles.

6.2 Completeness of the Framework to Find a Minimal Number

Here, we present a proof for the framework, i.e, a specification verified for a given set of requirements with a minimal

number of participants is also valid with any number of participants greater than the minimal number. First we discuss the restrictions specified below as Facts of our specification model that ensure a bound on the participant count for a given requirement.

Fact 5: *The specification can have a constraint specifying the number of required members for a role.*

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

Fact 6: *A role membership related condition can only refer to roles defined in the scope of ancestor activities.*

As there is always a fixed number of roles in ancestor activities, for the role model, there is a bound on the number of *invoker credentials*. For example, the *invoker credentials* for the *Checker* role are $\{Grader, Assistant\}$ and $\{Grader, Instructor\}$. That is an invoker has to be a member of either both the *Grader* and *Assistant* roles or both the *Grader* and *Instructor* roles to be a member of the *Checker* role. For the *Candidate* role, there is only one invoker-credentials, i.e., $\{Creator, Examinee, Student\}$. For the *Approver* role, there are two *invoker credentials*, $\{Adm2\}$ and $\{Instructor\}$.

Fact 7: *An event based condition can only refer to operations defined within the same activity scope.*

Due to Fact 5, we can enumerate all *invoker credentials* for a role operation within an activity.

Fact 8: Information can only flow through storage channels that are shared objects or information objects.

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

Based on the Fact 5, 6, 7, and 8, for a given requirement there is a bounded number of *invoker credentials* for all the role operations. There are more than one assignments of users in *initial assignment roles* to ensure execution paths in the model checker covers the *invoker credentials* for a requirement. As participants are assigned to roles in non-deterministic steps, this assignment ensures all possible *invoker credentials* for the role model. If the minimal number for the requirement is less than the required number as expressed by Fact 5, the model checker either reports an unreachable state showing that an operation cannot be executed or provides a trace showing that a larger number of user is required to satisfy the requirement.

On the other hand, we need to shown that for a specific requirement, the number of *invoker credentials* for the derived minimal number, remains same for any larger number of participants. Given that the derived minimal number is N with the equivalence class C, for a specific requirement of an activity template T. For K ($K \geq N$) number of users participate in an instance of T, assume that the set of *invoker credentials* is C. We need to show that the set C remains same when K+1 users participate in an instance of T. As

the the framework to find a minimal number only depend on the specification, the increment of participant count does not have any effect on the minimal number. Based on induction, as the *invoker credentials* set C remains same for any number of participants larger than N . As there exist a minimal number of participants for each specific types of requirement presented in [1], there exist a minimal number of participants for a set of requirements. Conversely, If there exist minimal number for all the requirements than there exists, then the number is also minimal for each of the requirements.

7. REFERENCES

- [1] T. Ahmed and A. R. Tripathi. Static Verification of Security Requirements in Role Based CSCW Systems. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 196–203, New York, June 2003. ACM.
- [2] 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.
- [3] 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 *Proceedings of International Conference on Software Engineering*, pages 439 – 448, 2000.
- [4] M. Corts and P. Mishra. DCWPL: a programming language for describing collaborative work. In *Proceedings of CSCW'96*, pages 21 – 29, November 1996.
- [5] R. Eshuis and R. Wieringa. Verification Support for Workflow Design with UML Activity Graphs. In *Proceedings of International Conference on Software Engineering*, pages 166 – 176, New York, 2002. ACM.
- [6] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [7] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying Business Processes using Spin. In *Proceedings of 4th International SPIN Workshop*, 1998.
- [8] D. Li and R. Muntz. COCA: Collaborative Objects Coordination Architecture. In *Proceedings of CSCW'98*, pages 179–188, 1998.
- [9] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proceedings 11th IEEE Computer Security Foundations Workshop* , pages 96 –105, June 1998.
- [10] P. Maggi and R. Sisto. Using SPIN to Verify Security Protocols. In *Proceedings of 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318*, pages 187–204, 2002.
- [11] P. Roberts and J.-P. Verjus. Towards Autonomous Descriptions of Synchronization Modules. In *Proceedings of IFIP Congress*, pages 981–986, Amsterdam, 1977. North-Holland.
- [12] A. Tripathi, T. Ahmed, and R. Kumar. Specification of Secure Distributed Collaboration Systems. In *IEEE International Symposium on Autonomous Distributed Systems*, pages 149–156, Los Alamitos, CA, April 2003. IEEE Computer Society Press.
- [13] A. Tripathi, T. Ahmed, R. Kumar, and S. Jaman. Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In *Proceedings of International Conference on Distributed Computing Systems 2002*, pages 393 – 400, July 2002.
- [14] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings First IEEE Symp. on Logic in Computer Science*, pages 322–331, Los Alamitos, CA, 1986. IEEE Computer Society Press.