Authorization Policy Specification and Enforcement for Group-Centric Secure Information Sharing

Ram Krishnan[1,2] and Ravi Sandhu[1]
[1]Institute for Cyber Security
[2]Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, TX

ram.krishnan@utsa.edu
ravi.sandhu@utsa.edu

Abstract. In this paper, we propose a methodology for incremental security policy specification at varying levels of abstraction while maintaining strict equivalence with respect to authorization state. We specifically consider the recently proposed group-centric secure information sharing (g-SIS) domain. The current specification for g-SIS authorization policy is *stateless* in the sense that it solely focuses on specifying the precise conditions under which authorization can hold in the system while only considering the history of actions that have occurred. The stateless application policy has been specified using linear temporal logic. In this paper, we develop an enforceable specification that is *stateful* in the sense that it is defined using specific data structures that are maintained in each state so as to make authorization decisions. We show that the stateful specification is authorization equivalent to that of stateless. That is, in any state, authorization will hold in stateful if and only if it also holds in the stateless specification.

Keywords: Authorization, Enforcement, Equivalence, Security Policy

1 Introduction

A fundamental problem in access control is the consistency of specification and enforcement of authorization policies. A large body of literature focuses on either the specification of authorization policies or its enforcement independent of each other. Our focus in this paper is to bridge these two areas. Our application domain is the recently proposed model for group-centric secure information sharing or g-SIS [4, 6]. In g-SIS, users and objects are brought together in a group to promote sharing and collaboration. Users may join and leave and objects may be added and removed from the group. The join, leave, add and remove operations may have different authorization semantics as will be discussed later. A formal set of core properties that are required of all g-SIS specifications have been defined given the basic group operations of join, leave, add and remove. Further, a specification, called the π-system, has been formulated and proven to satisfy the core g-SIS properties.

The π-system specification is defined in a *stateless* manner using first-order linear temporal logic (FOTL). (FOTL differs from the familiar propositional linear temporal

logic [7] by incorporating predicates with parameters, constants, variables, and quantifiers.) Specifically, the π-system is not directly enforceable in the way it is specified because it does not define the data structures that need to be maintained in order to make authorization decisions. Instead, the FOTL characterization of the π-system simply specifies the sequence of actions that need to have occurred in the past in order for authorization to hold at any given state. Thus, for example, a stateless specification may specify that a user may access an object in a group in a particular state if and only if the user had joined the group in the past, the object has been added to the group in the past and both the user and object are current members in the group (that is, the user has not left and the object has not been removed). Note that such a characterization using FOTL does not specify how to enforce that policy. A *stateful* specification, on the other hand, specifies the data structures that need to be maintained in the system so that they can be inspected in each state and authorization decisions be made.

In this paper, we develop a stateful specification for the π-system and prove that this specification is *authorization equivalent* to the stateless π-system specification. That is, a user will be authorized to access an object in a group in the stateful π-system specification *if and only if* it is also the case in the stateless π-system specification.

The separation of stateless from the stateful specification has a number of important virtues. A security policy researcher developing the stateless specification is not distracted by the data structures that need to be designed and maintained. Instead, she can focus purely on the precise characterization of the conditions under which authorization should hold in her system. Formal specification using FOTL also allows one to conduct rigorous formal analysis using automated techniques such as model checking as demonstrated in [4]. Once the stateless specification is developed, one can then focus on the data structure design and mechanisms needed to enforce the stateless policy. As will be shown, while the stateless specification may be complex for a non-expert in the field, the stateful specification is understandable and can be implemented by relatively competent programmers. The techniques we use include algorithmic specification of stateful π-system and induction for our proofs. We believe that this can be applied in various other application domains in which new policy specifications are developed.

This line of work is inspired in part by the relationship between the non-interference [2] and the Bell-LaPadula model [1]. The Bell-LaPadula model provides a lattice structure of security labels and the famous simple-security and star-properties to enforce one-directional information flow in the lattice. This is a stateful specification in that it describes data structures and rules that are enforceable. The non-interference specification is stateless and makes reference only to input-output behavior of a secure system. Our goals in this paper are to formalize authorization policy rather than information flow policy. Nonetheless the stateless and stateful distinction has strong similarities and the non-interference work has been inspirational. To the best of our knowledge, this is the first effort towards bridging authorization policy specification and enforcement.

The rest of the paper proceeds as follows. In section 2, we give a brief background on g-SIS and an overview of the stateless π-system specification. In section 3, we present a stateful specification for the π-system. In section 4, we show the equivalence of the stateful and stateless π-system specifications. We discuss future work and conclude in section 5.

2 Background

In this section, we provide a brief overview of g-SIS. A detailed discussion can be found in [4] and [6].

2.1 Overview of g-SIS

Future/Past	Operator	Read as	Explanation
Future	$\bigcirc$	Next	($\bigcirc$ p) means that the formula p holds in the next state.
	$\Box$	Henceforth	($\Box$ p) means that the formula p will continuously hold in all future states starting from the current state.
	$\mathcal{W}$	Unless	It says that p holds either until the next occurrence of q or if q never occurs, it holds throughout.
Past	$\blacklozenge$	Once	($\blacklozenge$ p) means that formula p held at least once in the past.
	$\mathcal{S}$	Since	(p $\mathcal{S}$ q) means that q happened in the past and p held continuously from the position following the last occurrence of q to the present.

Table 1. Intuitive summary of temporal operators used in this paper

In g-SIS, users may join, leave and re-join the group. Similarly, objects may be added, removed and re-added to the group. Authorization may hold in any state depending on the relative membership status of the user and object in question. The group operations join, leave, add and remove can be of different types with various authorization semantics. We use the following shorthand to denote such different semantics of group operations:

$$\begin{aligned}
\mathrm{Join}(u,g) &= (\mathrm{join}_1(u,g) \vee \mathrm{join}_2(u,g) \vee ... \vee \mathrm{join}_m(u,g)) \\
\mathrm{Leave}(u,g) &= (\mathrm{leave}_1(u,g) \vee \mathrm{leave}_2(u,g) \vee ... \vee \mathrm{leave}_n(u,g)) \\
\mathrm{Add}(o,g) &= (\mathrm{add}_1(o,g) \vee \mathrm{add}_2(o,g) \vee ... \vee \mathrm{add}_p(o,g)) \\
\mathrm{Remove}(o,g) &= (\mathrm{remove}_1(o,g) \vee \mathrm{remove}_2(o,g) \vee ... \vee \mathrm{remove}_q(o,g))
\end{aligned}$$

Thus, for instance, $\mathrm{join}_1(u,g)$ could represent a specific type of join operation that is different in authorization semantics from that of $\mathrm{join}_2(u,g)$. However, $\mathrm{Join}(u,g)$ captures the notion that a join operation of some type has occurred for u in g.

Definition 1 (State in Stateless Specification). *A state in the stateless specification is an interpretation that maps each predicate in the language to a relation over appropriate carriers.*

The predicates in the g-SIS language include action predicates such as Join, Leave, Add and Remove and an authorization predicate Authz. These predicates are specified over appropriate sorts (types). The semantic values over which a variable ranges depend on the variable's sort and are drawn from a set that is called the *carrier* of that sort. We use standard upper-case roman characters such as U (user sort) to denote sorts and calligraphic letters such as $\mathcal{U}$ (user carrier) to denote the corresponding carriers. A detailed discussion of the g-SIS language can be found in [4].

Definition 2 (Stateless Trace). *A trace in the stateless specification is an infinite sequence of states.*

The formulas that we specify below talk about stateless traces.

Well-Formed Traces We now introduce four formulas that define what we call *well-formed* g-SIS traces. (An intuitive overview of temporal operators used in this paper is provided in table 1.) The formulas we consider treat the authorization a user has to access an object independently of actions involving other users and objects. Thus, from here on it is often convenient to omit the parameters in all of the predicates. We also omit the quantifiers as they can be easily inferred from the context (join and leave are user operations, add and remove are object operations).

A. An object cannot be Added and Removed and a user cannot Join and Leave at the same time.[1]

$$\tau_0 = \Box(\neg(\mathrm{Add} \wedge \mathrm{Remove}) \wedge \neg(\mathrm{Join} \wedge \mathrm{Leave}))$$

B. For any given user or object, two types of operations cannot occur at the same time.

$$\begin{aligned}\tau_1 = &\forall i,j\ \Box((i \neq j) \rightarrow \neg(\mathrm{join}_i \wedge \mathrm{join}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\mathrm{leave}_i \wedge \mathrm{leave}_j)) \wedge \\ &\forall i,j\ \Box((i \neq j) \rightarrow \neg(\mathrm{add}_i \wedge \mathrm{add}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\mathrm{remove}_i \wedge \mathrm{remove}_j))\end{aligned}$$

Thus, for example, a user cannot join with 2 different semantics in the same state. Multiple occurrences of the same event in a given state (i.e. when i equals j above) are treated as a single occurrence of that event in FOTL.

C. If a user u joins a group, u cannot join again unless u first leaves the group. Similar rules apply for other operations.

$$\begin{aligned}\tau_2 = &\Box(\mathrm{Join} \rightarrow \bigcirc\,(\neg\mathrm{Join}\ \mathcal{W}\ \mathrm{Leave})) \wedge \Box(\mathrm{Leave} \rightarrow \bigcirc\,(\neg\mathrm{Leave}\ \mathcal{W}\ \mathrm{Join})) \wedge \\ &\Box(\mathrm{Add} \rightarrow \bigcirc\,(\neg\mathrm{Add}\ \mathcal{W}\ \mathrm{Remove})) \wedge\ \Box(\mathrm{Remove} \rightarrow \bigcirc\,(\neg\mathrm{Remove}\ \mathcal{W}\ \mathrm{Add}))\end{aligned}$$

D. A Leave event cannot occur before Join. Similarly for objects.

$$\tau_3 = \Box(\mathrm{Leave} \rightarrow \blacklozenge\mathrm{Join}) \wedge \Box(\mathrm{Remove} \rightarrow \blacklozenge\mathrm{Add})$$

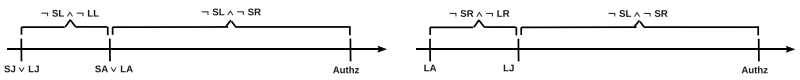
Thus, in any given trace, an object needs to be added before a remove operation may occur in any state.

2.2 The Stateless π-system g-SIS Specification

The π-system specification supports two types of semantics for join, leave, add and remove operations namely: strict and liberal.

A strict join (SJ) allows the joining user to access only those objects added on or after the state in which the user joins. A liberal join (LJ), in addition, allows the joining user to access objects added to the group prior to the join state.

[1] Note that here and below we introduce names of the form τ_j for each of the formulas for later reference. The equality introduces shorthands for the respective formulas.



(a) Formula λ_1. (b) Formula λ_2.

Fig. 1. Stateless specification illustration.

On strict leave (SL), the user loses access to all objects in the group. On liberal leave (LL), the user retains access to all objects that were authorized in the leaving state.

Similarly, for objects, on strict add (SA), the added object may be accessed only by users who have joined at or prior to the state in which the object is added to the group. Liberal add (LA) does not have such a constraint.

On strict remove (SR), the object cannot be accessed by any user. On liberal remove (LR), the object may be accessed by users who were authorized to access the object in the remove state.

The π-system specification supports the strict and liberal semantics for group operations. Given that different users may join and leave with different semantics and different objects may be added and removed with different semantics, the π-system specifies the precise conditions under which authorization for a user to access an object in a group may hold in the system.

Definition 3 (Stateless π-system). *The stateless π-system specification, $\pi_{stateless}$, accepts traces satisfied by the following formula:*

$$\forall u.\forall o.\forall g.\Box(\mathrm{Authz}(u,o,g,\mathrm{read}) \leftrightarrow \ \lambda_1 \vee \lambda_2) \wedge \bigwedge_{0 \leq j \leq 3} \tau_j$$

where,

$$\lambda_1 = ((\neg\mathrm{SL} \wedge \neg\mathrm{SR})\ \mathcal{S}\ ((\mathrm{SA} \vee \mathrm{LA}) \wedge ((\neg\mathrm{LL} \wedge \neg\mathrm{SL})\ \mathcal{S}\ (\mathrm{SJ} \vee \mathrm{LJ}))))$$
$$\lambda_2 = ((\neg\mathrm{SL} \wedge \neg\mathrm{SR})\ \mathcal{S}\ (\mathrm{LJ} \wedge ((\neg\mathrm{SR} \wedge \neg\mathrm{LR})\ \mathcal{S}\ \mathrm{LA})))$$

and the τ_j's are the well-formedness constraints.

Given a specific user and an object, note that formula λ_1 handles the scenario where an add operation occurs after a join operation (figure 1(a)) and formula λ_2 handles the scenario where an add operation occurs before a join operation (figure 1(b)). (Here, due to the semantics of the strict add and strict join, we do not need to check for their occurrence in formula λ_2 illustrated in figure 1(b)). In [6], we have shown that the specification above is consistent with the semantics of strict and liberal operations discussed earlier. In addition, we have specified a set of core security properties that are required of any g-SIS specification and shown that the stateless π-system specification discussed above satisfies those core properties.

A g-SIS stateless specification with read and write operations (that supports multiple versions of the object) has been specified in [4]. Although we consider a stateless specification for read authorization in this paper, our discussion is not specific to the type of permission.

```
main(){
    // Phase 1 and 2 time periods below are allocated such that phase 1 occurs before
    // phase 2 and tasks in perTick step below conclude before the tick interval elapses.
    perTick: During each tick interval i {
        Phase 1:{ // Steps 1.1, 1.2 and 1.3 may execute concurrently.
            1.1. For each user, accept the first request received and
            store that information in variable userReq(u,g).
            // the request received could be one of:
            // SJReq(u,g), LJReq(u,g), SLReq(u,g) and LLReq(u,g).
            1.2. For each object, accept the first request received and
            store that information in variable objectReq(o,g).
            // the request received could be one of:
            // SAReq(o,g), LAReq(o,g), SRReq(o,g) and LRReq(o,g).*/
            1.3. Accept all the authorization requests:
                if (isAuthz(u,o,g)) authzReq=authzReq ∪ isAuthz(u,o,g)
                // isAuthz(u,o,g) represents authorization request for user u to access object o.
        }
        Phase 2:{ // Steps 2.1 and 2.2 must be sequential. However, the processing of
            // captured requests in step 2.1 may be done concurrently.
            2.1. For each captured request, invoke the corresponding function in
            table 3 with the appropriate parameters.
            // for example, if userReq(u,g) is SJReq(u,g), invoke userEvent(u,g,join,i,strict).
            2.2. Process each authorization request:
                for each (isAuthz(u,o,g) ∈ authzReq)
                    authzResult(u,o,g)=authzSF(u,o,g);
        }
        Reset all variables appropriately.
    }
}
```

Table 2. Stateful Specification (Request Handling)

3 Stateful π-system

In this section, we develop a stateful π-system specification that is authorization equivalent to the stateless specification—that is a user will be authorized to access an object in the stateful system if and only if it is also the case in the stateless system. Evidently, the stateless specification is highly abstract and specified using FOTL. The stateful specification that we develop is an incremental step in the direction of a concrete implementation of a system that is reasonably authorization equivalent to the stateless specification. We say "reasonably" because it is our fundamental hypothesis that all practical distributed systems will inevitably face real-world issues such as network delay and caching, which will lead to authorization inconsistencies with the idealized stateless specification. Thus such systems can at most be approximate to the stateless specification. One such approximation is the notion of stale-safety [3] that bounds acceptable delay between the time at which an action (such as reading an object) was known to be authorized and the time at which that action is actually performed. Our future refinements of the stateful π-system will consider various notions of such approximations.

```
int userEvent(User u, Group g, uEvent e, interval t, uSemantics s){
      1. Check that the current uEvent e is not the same as the
      uEvent value in the previous tuple in table(u,g). If so, return 0.
      // This ensures, for example, that a join event is not followed
      // immediately by another join.
      2. Also check, in case the table is empty, then e is not an SL or LL. If so, return 0.
      // This ensures that the first user event entry in table(u,g) is not leave.
      3. Enter <t,e,s> into table(u,g) and return 1.
}
int objectEvent(Object o, Group g, oEvent e, interval t, oSemantics s){
      1. Check that the current oEvent e is not the same as the
      oEvent value in the previous tuple in table(o,g). If so, return 0.
      // This ensures, for example, that an add event is not followed
      // immediately by another add.
      2. Also check, in case the table is empty, then e is not an SR or LR. If so, return 0.
      // This ensures that the first object event entry in table (o,g) is not remove.
      3. Enter <t,e,s> into table(o,g) and return 1.
}
```

Table 3. Stateful Specification (enforcing well-formedness constraints.)

As the first transition from an abstract specification towards an implementable specification, the stateful specification that we design is centralized in the sense that authorization decisions are made based on data structures maintained in a specific repository for each user and object. There could be different repositories for different users and objects that may be distributed on the whole. Specifically, we are not concerned about replication of data structures of a user or an object and maintenance of its consistency. We also not concerned about distributing parts of the data structure of a user or an object. Authorization decisions for a specific user to access a specific object are made based on their specific data structures maintained at specific repositories.

Note that the stateless specification simply does not admit traces of actions that do not obey the well-formedness constraints. More importantly, it does not (intentionally) specify how one should handle ill-formed traces. At the stateful specification level of abstraction, one must begin to address such issues. Many strategies may be employed—we will consider one for our stateful specification (discussed later).

3.1 Stateful π-system Design

In the stateful π-system, the data structures that we maintain and consult with for making authorization decisions are simple relations for users and objects in the group—which we refer to informally as tables. For instance, the data structure for a user u in a group g, table(u,g), contains a history of that user's joins and leaves in the group. (The group parameter g is specified for being precise. The reader may safely ignore this in the rest of this paper as we focus only on one group at any time.) The format of each tuple in table(u,g) is: <*time-stamp, event, semantics*>. Here *event* is either join or leave, *semantics* is either strict or liberal and *time-stamp* specifies the time at which this event occurred as per a global clock. Thus a snapshot of table(u,g) at any point in time

```
int authzSF(User u, Object o, Group g){

step 1: Fetch tables table(u,g) and table(o,g). If either table is empty, return 0.
        Merge sort table(u,g) and table(o,g) in ascending order of timestamp.
            In case of same timestamp, follow precedence rules apply:
              (i) Add and Join same timestamp: Add follows Join
              (ii) Join and Remove same timestamp: Join follows Remove
              (iii) Add and Leave same timestamp: Add follows Leave
              (iv) Remove and Leave same timestamp: any order
        Let n be the total number of entries in the merged table.

step 2: for i=1 to n{
          case event[i]=join{
step 2a:      (i) Step down the table looking for an add event. If a leave event is encountered
              prior to add event, continue step 2 for loop. If no add event found, return 0.
              (ii) From the point the add event was found in the table, step down all the way
              to index n ensuring no SL or SR is encountered.
              If SL found, continue step 2. If SR found, continue step 2a from current index.
              (iii) return 1;
          }
          case event[i]=add && eventType[i]=liberal{
step 2b:      (i) Step down the table looking for an LJ event. If a remove event is encountered
              prior to LJ event, continue step 2 for loop. If no LJ event found, return 0.
              (ii) From the point the LJ event was found in the table, step down all the way
              to index n ensuring no SL or SR is encountered.
              If SR found, continue step 2. If SL found, continue step 2b from current index.
              (iii) return 1;
          }
        }
step 3: return 0;
}
```

Table 4. Stateful Specification (Authorization Decision)

gives a chronological history of the user joining and leaving (possibly many times) and whether they were of strict or liberal type. Similarly, a tuple in an object data structure, table(o,g), has the same format as the user table except *event* is either add or remove. Note that the number of tuples in any table is not bounded.[2]

The stateful specification for the π-system is presented in tables 2, 3 and 4. The authzSF function in table 4 returns 1 if a user u is authorized to access an object o, 0 otherwise. It does so by inspecting the data structures: table(u,g) and table(o,g). As mentioned earlier, the stateful π-system must also specify how the requests to join, leave, add and remove and requests to ascertain if users are authorized to read objects

[2] Keeping them unbounded has many virtues. For instance, as we will see, this facilitates user data structures not being touched when an object data structure needs to be updated (and vice-versa). Of course, there are other data structure designs where they may be bounded but with different pros and cons.

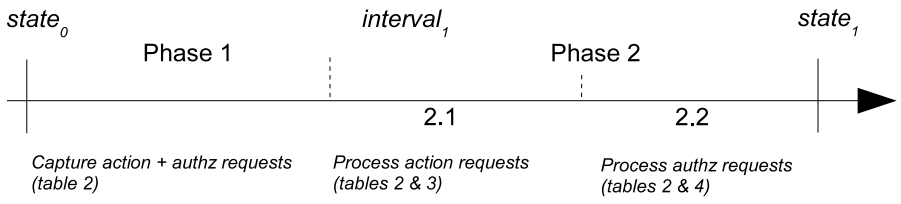


Fig. 2. Stateful π-system Overview.

are processed. Tables 2 and 3 specify one of many possible ways to do this. We discuss each of these 3 components of the stateful π-system in further detail below.

3.2 Stateful π-system Specification

An overview of how the functions in the tables 2, 3 and 4 interact is given in figure 2. Consider the main function in table 2. It receives and processes action requests (requests to join, leave, add and remove) and authorization requests during the time interval between any two clock ticks. The function works in two phases during each time interval. During phase 1, it receives the action and authorization requests. It filters the action requests so that only the first user request and the first object request are captured. (Different strategies for capturing action requests may be employed—e.g. it need not be the first request received that is captured.) This ensures, for instance, that only a join or a leave request of a specific type (strict or liberal) is captured for any given user but not both. However, all authorization requests are captured during phase 1. When phase 1 completes, further new requests are not admitted. During phase 2, first all action requests received in phase 1 are processed using the user and object event processing functions in table 3 and then all the captured authorization requests are evaluated using authzSF function in table 4. At the end of phase 2, the data structures are up-to-date and authorization decisions are complete for all the requests received in phase 1.

Consider the function userEvent in table 3 which processes the user requests received by the function in table 2. The check performed in step 1 ensures that user requests to repeatedly join without an intermittent leave (and vice-versa) are ignored. Similarly, step 2 ensures that the first entry in the table does not begin with a leave operation. If all is well, a new tuple is entered into the table and the function returns 1. The function returns 0 in all other cases. The objectEvent function similarly processes object requests. Note that tables 2 and 3 together achieve well-formedness constraints of stateless π-system specification.

The function authzSF in table 4 returns 1 if a user u is authorized to access an object o in group g, 0 otherwise. This algorithm can be optimized but we keep it straightforward for simpler presentation. It begins by taking the corresponding user and object tables as input. Note that if either table is empty (i.e., either the user or the object has never been a member of the group), the user is not authorized to read the object. By appending the tuples to the respective tables as the events occur, table(u,g) and table(o,g) are pre-sorted with respect to the time-stamp. The function merge sorts these

two tables based on the time-stamp entries to obtain a table of events of u and o in the chronological order of occurrence. In the event a user and object entry in the respective tables have the same time-stamp, we specify precedence rules to resolve the tie for sorting the tuples consistent with temporal operator semantics in the stateless π-system. If Add and Join occur at the same time, Add follows Join. If Join and Remove occur at the same time, Join follows Remove. If Add and Leave occur at the same time, Add follows Leave. Finally, if Remove and Leave occur at the same time, they can be merge sorted in any order. Let the total number of entries in the merged table be n.

The algorithm proceeds by iterating through each tuple in this new merge sorted table. We assume that event[i] fetches the specific event (such as join or add) from the i^{th} entry in the merged table and eventType[i] fetches the respective semantics (such as strict or liberal) of that event from the same tuple. Each of the two cases in the for loop looks for an overlapping period of authorizing membership between the user and object, much like formulas λ_1 and λ_2. The first case looks for a join event followed by an add event (see figure 1(a)) and the second case looks for an add event followed by a join event (see figure 1(b)). As per λ_2, the second case looks for a liberal add followed by a liberal join. The remaining part of the case statements conduct checks to ensure that there is no subsequent de-authorizing event such as strict leave or remove following this point of authorization. If there is none, the algorithm returns 1 indicating that the user is authorized. Otherwise it returns 0 after step 3.

3.3 Implementation Considerations

Evidently, the stateful specification that has been presented in tables 2, 3 and 4 can be comprehended and implemented by a competent programmer as compared to the temporal logic based stateless specification. Since the stateless specification has been analysed and certain security properties have been proven [4, 6] and has been shown to be authorization equivalent to the stateful specification (section 4), the stateful specification also is guaranteed to have those security properties.

As mentioned earlier, the authzSF function in table 4 is not designed for efficiency but for ease of presentation. The worst case time complexity of this function is roughly $\mathcal{O}(n^2)$ where n is the sum of the number of events in the user and object tables. This is because for each of the n iterations of the outer for loop in step 2, the loops in one of the inner case statements could run through a maximum of n iterations.

This stateful specification has a few limitations. For instance, both the user and object tables are unbounded. Nevertheless, this is not a major issue in many practical applications in which membership status of users and objects do not change frequently. Also, due to nature of phases 1 and 2 in table 2, all action requests need to be received before they can be processed. Thus during phase 2 of interval, no requests will be accepted. The ordering of tasks in two phases ensures that the requests received during the time interval will affect the authorization values that hold at the upcoming state. These constraints may be unacceptable for certain application scenarios. Addressing such limitations of the stateful specification is not the primary focus of this paper. Note that the current stateful specification design allows user and object data structures to be maintained in a distributed manner so that if a user membership status changes, it does not require updating data structures of other users and objects in the system. One can

design alternate stateful specifications for the same stateless specification with different trade-offs. For instance, one can maintain a single data structure that involves both users and objects. But changes in any object's group membership status will entail updating entries for all users in the system. This would have limitations in distributing it.

4 Equivalence of Stateful and Stateless π-system Specifications

In this section, we show that the stateful specification is authorization equivalent to the stateless specification. That is, in all possible traces, a user will be authorized to access an object at any given state in the stateful π-system if and only if it is also the case in the stateless π-system.

Given that we are dealing with traces in the stateless specification, we propose a similar notion of traces in the stateful specification.

Definition 4 (State in Stateful Specification). *A state in the stateful specification is a specific interpretation of every user and object data structure maintained in the system at the end of every clock tick.*

Definition 5 (Stateful Trace). *A trace in the stateful specification is an infinite sequence of states.*

Definition 6 (Stateful π-system). *The stateful π-system specification, $\pi_{stateful}$, is given in table 2 which consists of functions from tables 3 and 4.*

Our goal now is to show that given a stateless and a corresponding stateful trace, authorization is equivalent in every state. To establish this "correspondence", we specify mappings that would take a stateless trace and create a stateful trace and vice-versa.

Notation We use σ to denote a stateless trace and $\widehat{\sigma}$ to denote a stateful trace. σ_i refers to state i in a trace σ with infinite states. We use $\sigma_{i,j}$ to denote a state i in σ where we only consider the first j states. Actions are represented using relations. Thus $\langle \mathbf{u}, \mathbf{g} \rangle \in [\![\mathrm{SJ}_{stateless}]\!]\sigma_i$ denotes that a user $\mathbf{u}$ is strictly joined to group $\mathbf{g}$ in state i in a stateless trace σ. Similarly, $\langle i, \mathrm{Join}, \mathrm{Liberal} \rangle \in [\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i$ denotes user $\mathbf{u}$ has liberally joined group $\mathbf{g}$ in state i in a stateful trace $\widehat{\sigma}$.

Note that the time interval that a clock tick corresponds to is abstract. Any event request (such as a request to join) that is processed during a transition from clock tick (state) i to i+1 will receive a time-stamp of i+1. This convention makes stateful specification consistent with the FOTL semantics in the stateless specification.

Definition 7 (Action Trace). *Given a stateless or stateful trace in the π-system, an* action trace *is a sequence of states excluding the authorization relation.*

Definition 8 (Action Equivalence). *A stateful trace $\widehat{\sigma}$ and a stateless trace σ are* action equivalent *if the join, leave, add and remove actions match for every user and object in every group in the corresponding states in $\widehat{\sigma}$ and σ.*

Definition 9 (α-mapping). *Given a stateless trace σ in $\pi_{stateless}$, α-mapping creates an action equivalent stateful trace $\widehat{\sigma}$ in $\pi_{stateful}$.*

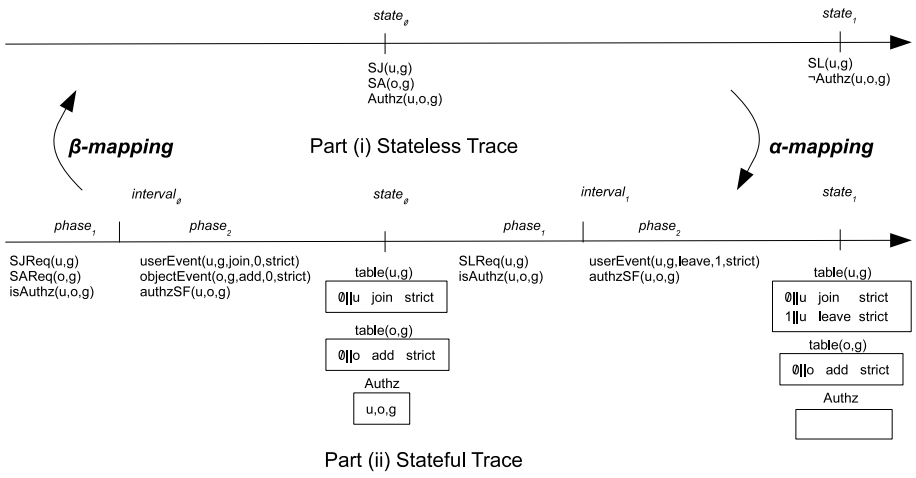


Fig. 3. α and β mapping. Part (i) shows a sample stateless trace and part (ii) shows a corresponding stateful trace. Note that the stateful trace generates the required action and authorization requests during the time interval leading up to the state.

Rules used for α-mapping are straight-forward and given here by example. For example (see figure 3), for each $\langle \mathbf{u}, \mathbf{g} \rangle \in [\![\mathrm{SJ}_{stateless}]\!]\sigma_i$, create an entry $\langle i, \mathrm{Join}, \mathrm{Strict} \rangle$ in $[\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i$. This is achieved by sending a SJReq(**u**,**g**) (see table 2) during phase 1 in the time interval between the state transition from $\widehat{\sigma}_{i-1}$ to $\widehat{\sigma}_i$. Similarly, for each $\langle \mathbf{u}, \mathbf{g} \rangle \in [\![\mathrm{LJ}_{stateless}]\!]\sigma_i$, create an entry $\langle i, \mathrm{Join}, \mathrm{Liberal} \rangle$ in $[\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i$. Similar rules apply to other predicates.

Definition 10 (**β-mapping).** *Given a stateful trace $\widehat{\sigma}$ in $\pi_{stateful}$, β-mapping creates an action equivalent stateless trace σ in $\pi_{stateless}$.*

Rules used for β-mapping are straight-forward and given here by example. For example (see figure 3), for each tuple in $[\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i - [\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_{i-1}$, create that entry in corresponding relation in the stateless trace. That is if $\langle i, \mathrm{Join}, \mathrm{Strict} \rangle \in [\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i - [\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_{i-1}$, then create an entry $\langle \mathbf{u}, \mathbf{g} \rangle$ in $[\![\mathrm{SJ}_{stateless}]\!]\sigma_i$. Similarly, for each $\langle i, \mathrm{Join}, \mathrm{Liberal} \rangle \in [\![\mathrm{table}(\mathbf{u}, \mathbf{g})]\!]\widehat{\sigma}_i$, create an entry $\langle \mathbf{u}, \mathbf{g} \rangle$ in $[\![\mathrm{LJ}_{stateless}]\!]\sigma_i$. Similar rules apply to other operations in the stateful specification.

Lemma 1. *For every action trace σ that is generated by $\pi_{stateless}$, a stateful action trace $\widehat{\sigma}$ constructed using α-mapping is accepted by $\pi_{stateful}$.*

By the term "accepted by" above, we mean that by inputting an α-mapped trace to the stateful π-system, the data structure it maintains must reflect the exact action trace of the stateless π-system (see figure 3 for example).

Lemma 2. *For every action trace $\widehat{\sigma}$ generated by $\pi_{stateful}$, a stateless action trace constructed using β-mapping is accepted by $\pi_{stateless}$.*

By the term "accepted by" above, we mean that the β-mapped stateless action trace will be well-formed as per the stateless π-system specification. The proofs of lemmas 1 and 2 follow directly from their definitions. Due to space limitations, the proofs are provided in [5]. Next, we have the following 2 lemmas.

Lemma 3 (Soundness). *For every trace $\widehat{\sigma}$ accepted by $\pi_{stateful}$, there exists a β-mapped trace σ that is accepted by $\pi_{stateless}$ such that:*

$$\forall i \in \mathbb{N}. \forall t \in \langle \mathcal{U}, \mathcal{O}, \mathcal{G} \rangle. t \in [\![\mathrm{Authz}_{\pi_{stateful}}]\!]\widehat{\sigma}_i \rightarrow t \in [\![\mathrm{Authz}_{\pi_{stateless}}]\!]\sigma_i$$

Lemma 4 (Completeness). *For every trace σ accepted by $\pi_{stateless}$, there exists an α-mapped trace $\widehat{\sigma}$ that is accepted by $\pi_{stateful}$ such that:*

$$\forall i \in \mathbb{N}. \forall t \in \langle \mathcal{U}, \mathcal{O}, \mathcal{G} \rangle. t \in [\![\mathrm{Authz}_{\pi_{stateless}}]\!]\sigma_i \rightarrow t \in [\![\mathrm{Authz}_{\pi_{stateful}}]\!]\widehat{\sigma}_i$$

Due to space limitations, the proofs for lemmas 3 and 4 are provided in [5]. The proofs are inductive.

Theorem 1. *The stateful and stateless π-system specifications are* authorization equivalent*. That is:*

$$\forall i \in \mathbb{N}. \forall t \in \langle \mathcal{U}, \mathcal{O}, \mathcal{G} \rangle. t \in [\![\mathrm{Authz}_{\pi_{stateful}}]\!]\widehat{\sigma}_i \leftrightarrow t \in [\![\mathrm{Authz}_{\pi_{stateless}}]\!]\sigma_i$$

Proof. The theorem follows from lemmas 3 and 4.

The above theorem states that in every state in a stateful trace, the authorization relation is equivalent to that of the corresponding state in a statefless trace. We have shown that a highly abstract temporal logic based stateless specification can be grounded in a concrete stateful specification while maintaining equivalency with respect to authorization.

5 Conclusion and Future Work

We presented a methodology for consistent specification and enforcement of authorization policies. The stateless specification is highly conducive to automated formal analysis using techniques such as model checking. However, it cannot be enforced using the way it is specified. The stateful specification focuses on how to enforce the stateless policy using distributed data structures and associated algorithms. This specification can be implemented by programmers. We have established a formal bridge between a highly abstract stateless specification and a relatively concrete stateful specification. The next level of refinement is to generate distributed specification which account for approximation (for example, due to network delay and caching) with respect to stateless specification and a concrete implementation. Such incremental refinement of policy specification while maintaining consistency at each transition is critical in secure systems design.

Our current stateful specification, although highly distributed, maintains unbounded history of user and object actions. Our follow on work focuses on generating alternate stateful specifications. One approach is to generate a stateful specification with bounded

data structures that maintain information about authorization status of each user for each object. While this bounds the data structures, it requires modifying every object's data structures if the user's membership status changes in the group. Another approach is to generate a hybrid specification that combine the pros and cons of the two approaches above and proving equivalence with respect to stateless specification. We believe our methodology can be extended to other application domains with suitable adaptation of proof techniques as needed.

Acknowledgments

The authors are partially supported by grants from AFOSR MURI and State of Texas Emerging Technology Fund.

References

1. Bell, D., La Padula, L.: Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306 (1975)
2. Goguen, J., Meseguer, J.: Security policies and security models. IEEE Symposium on Security and Privacy (1982)
3. Krishnan, R., Niu, J., Sandhu, R., Winsborough, W.: Stale-safe security properties for group-based secure information sharing. In: Proceedings of the 6th ACM workshop on Formal methods in security engineering. pp. 53–62. ACM New York, NY, USA (2008)
4. Krishnan, R., Niu, J., Sandhu, R., Winsborough, W.: Group-centric secure information sharing models for isolated groups. To appear in ACM Transactions on Information and Systems Security (2011). Camera ready copy available at: `http://engineering.utsa.edu/~krishnan/journals/2010-tissecSACMAT.pdf`
5. Krishnan, R., Sandhu, R.: Authorization Policy Specification and Enforcement for Group-Centric Secure Information Sharing *(Full Version)*. Tech. Rep. CS-TR-2011-016, University of Texas at San Antonio, September 2011. Also available at: `http://engineering.utsa.edu/~krishnan/conferences/2011iciss-full.pdf`
6. Krishnan, R., Sandhu, R., Niu, J., Winsborough, W.H.: Foundations for group-centric secure information sharing models. In: Proc. of ACM symposium on access control models and technologies (2009)
7. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science. pp. 46–67 (1977)