Proceedings of IEEE Computer Security Foundations Workshop VI, Franconia, NH, June 15-17, 1993, pages 109-118.

On Testing for Absence of Rights in Access Control Models

*Ravi S. Sandhu and Srinivas Ganta**

Center for Secure Information Systems, and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030

Abstract

The well-known access control model formalized by Harrison, Ruzzo, and Ullman (HRU) does not allow testing for absence of access rights in its commands. Sandhu's Typed Access Matrix (TAM) model, which introduces strong typing into the HRU model, continues this tradition. Ammann and Sandhu have recently proposed an extension of TAM called augmented TAM (ATAM), which allows testing for absence of rights. The motivation for ATAM is to express policies for dynamic separation of duties based on transaction control expressions.

In this paper we study the question of whether or not testing for absence of access rights adds fundamental expressive power. We show that TAM and ATAM are formally equivalent in their expressive power. However, our construction indicates that while testing for absence of rights is theoretically unnecessary, such testing appears to be practically beneficial.

1 Introduction

The need for access controls arises in any computer system that provides for controlled sharing of information and other resources among multiple users. Access control models (also called protection models or security models) provide a formalism and framework for specifying, analyzing and implementing security policies in multi-user systems. These models are typically defined in terms of the well-known abstractions of subjects, objects and access rights; with which we assume the reader is familiar.

Access controls are useful to the extent they meet the user community's needs. They need to be flexible so that individual users can specify access of other users to the objects they control. At the same time the discretionary power of individual users must be constrained to meet the overall objectives and policies of an organization. One method for achieving the desired flexibility is to allow security administrators to specify policies for propagation of rights, which allow discretionary freedom to users but at the same time impose non-discretionary rules. Several such policies, and access control models for their specification, have been published in the literature (see, for example, any of the references cited in this paper).

The access control model formalized by Harrison, Ruzzo, and Ullman (HRU) [3] was the first model to propose a language for allowing security administrators to specify their security policy in terms of propagation of access rights. A significant characteristic of the HRU model is that it does not allow testing for absence of rights. The take-grant model [4], and its variations [9], similarly do not allow such testing. Sandhu's Typed Access Matrix (TAM) model [7, 8], which introduces strong typing into HRU, continues this tradition.

In contrast to HRU and TAM, Ammann and Sandhu [2] have recently proposed an extension of TAM, called augmented TAM (ATAM), which allows testing for absence of rights. The motivation for developing ATAM is to express policies for dynamic separation of duties based on transaction control expressions [6]. Such policies have a natural, and relatively straightforward, expression based on testing for absence of access rights.

In this paper we investigate the question of whether or not testing for absence of rights adds fundamental expressive power to access control models. Our principal contribution is to demonstrate that TAM and ATAM are formally equivalent in expressive power. However, the nature of our construction indicates that even though testing for absence of rights is theoreti-

*The work of both authors is partially supported by National Science Foundation grant CCR-9202270. The authors are grateful to Nathaniel Macon for his support and encouragement in making this work possible.

cally unnecessary, such testing appears to be practically beneficial.

The rest of the paper is organized as follows. Section 2 gives a brief review of TAM and ATAM. Section 3 proves formal equivalence of the expressive power of TAM and ATAM, by showing how any given ATAM system can be simulated by a TAM system. Section 4 gives our conclusions, including an informal discussion of why our construction indicates that testing for absence of rights could be practically beneficial.

2 Background

In this section we review the definition of TAM, which was introduced by Sandhu in [7]. Our review is necessarily brief. The motivation for developing TAM, and its relation to other access control models are discussed at length in [7]. Following the review of TAM we briefly review the definition of ATAM [2].

2.1 The Typed Access Matrix (TAM) Model

The principal innovation of TAM is to introduce strong typing of subjects and objects, into the access matrix model of Harrison, Ruzzo and Ullman [3]. This innovation is adapted from Sandhu's Schematic Protection Model [5], and its extension by Ammann and Sandhu [1].

As one would expect from its name, TAM represents the distribution of rights in the system by an access matrix. The matrix has a row and a column for each subject, and a column for each object. Subjects are also considered to be objects. The $[X, Y]$ cell contains rights which subject X possesses for object Y.

Each subject or object is created to be of a specific type, which thereafter cannot be changed. It is important to understand that the types and rights are specified as part of the system definition, and are not predefined in the model. The *security administrator* specifies the following sets for this purpose:

- a finite set of access *rights* denoted by R, and
- a finite set of *object types* (or simply *types*), denoted by T.

For example, $T = \{user, so, file\}$ specifies there are three types, viz., user, security-officer and file. A typical example of rights would be $R = \{r, w, e, o\}$ respectively denoting read, write, execute and own. Once these sets are specified they remain fixed, until the security administrator changes their definition. It should be kept in mind that TAM treats the security administrator as an external entity, rather than as another subject in the system.

The *protection state* (or simply *state*) of a TAM system is given by the four-tuple (OBJ, SUB, t, AM) interpreted as follows:

- OBJ is the set of *objects*.
- SUB is the set of *subjects*, $SUB \subseteq OBJ$.
- $t : OBJ \rightarrow T$, is the *type function* which gives the type of every object.
- AM is the *access matrix*, with a row for every subject and a column for every object. The contents of the $[S, O]$ cell of AM are denoted by $AM[S, O]$. We have $AM[S, O] \subseteq R$.

  For convenience we usually drop the prefix AM, and understand $[S, O]$ to denote $AM[S, O]$.

The rights in the access matrix cells serve two purposes. First, presence of a right, such as $r \in [X, Y]$ may authorize X to perform, say, the read operation on Y. Second, presence of a right, say $o \in [X, Y]$ may authorize X to perform some operation which changes the access matrix, e.g., by entering r in $[Z, Y]$. In other words, X as the owner of Y can change the matrix so that Z can read Y.

The protection state of the system is changed by means of TAM commands. The security administrator defines a finite set of TAM commands when the system is specified. Each TAM *command* has one of the following formats.

> **command** $\alpha(X_1 : t_1,\ X_2 : t_2,\ \ldots,\ X_k : t_k)$
>   **if** $r_1 \in [X_{s_1}, X_{o_1}] \wedge r_2 \in [X_{s_2}, X_{o_2}] \wedge \ldots$
> $\wedge\ r_m \in [X_{s_m}, X_{o_m}]$
>   **then** $op_1;\ op_2;\ \ldots;\ op_n$
> **end**

or

> **command** $\alpha(X_1 : t_1,\ X_2 : t_2,\ \ldots,\ X_k : t_k)$
> $op_1;\ op_2;\ \ldots;\ op_n$
> **end**

Here α is the *name* of the command; X_1, X_2, ..., X_k are *formal parameters* whose types are respectively t_1, t_2, ..., t_k; r_1, r_2, ..., r_m are rights; and s_1, s_2,

..., s_m and o_1, o_2, ..., o_m are integers between 1 and k. Each op_i is one of the *primitive operations* discussed below. The predicate following the **if** part of the command is called the *condition* of α, and the sequence of operations op_1; op_2; ...; op_n is called the *body* of α. If the condition is omitted the command is said to be an *unconditional command*, otherwise it is said to be a *conditional command*.

A TAM command is invoked by substituting actual parameters of the appropriate types for the formal parameters. The condition part of the command is evaluated with respect to its actual parameters. The body is executed only if the condition evaluates to true.

There are six primitive operations in TAM, grouped into two classes, as follows.

- *Monotonic Primitive Operations*
  enter r **into** $[X_s, X_o]$
  create subject X_s **of type** t_s
  create object X_o **of type** t_o

- *Non-Monotonic Primitive Operations*
  delete r **from** $[X_s, X_o]$
  destroy subject X_s
  destroy object X_o

It is required that s and o are integers between 1 and k, where k is the number of parameters in the TAM command in whose body the primitive operation occurs.

The **enter** operation enters a right $r \in R$ into an existing cell of the access matrix. The contents of the cell are treated as a set for this purpose, i.e., if the right is already present the cell is not changed. The **enter** operation is *monotonic*, because it only adds and does not remove from the access matrix. The **delete** operation has the opposite effect of **enter**. It (possibly) removes a right from a cell of the access matrix. Since each cell is treated as a set, **delete** has no effect if the deleted right does not already exist in the cell. Because **delete** (potentially) removes a right from the access matrix, it is a *non-monotonic* operation.

The **create subject** and **destroy subject** operations make up a similar monotonic versus non-monotonic pair. The **create subject** operation requires that the subject being created has a unique identity different not only from existing subjects, but also different from all subjects that have ever existed thus far. The **destroy subject** operation requires that the subject being destroyed currently exists. Note that if the pre-condition for any **create** or **destroy** operation in the body is false, the entire TAM command has no effect. The **create subject** operation introduces an empty row and column for the newly created subject into the access matrix. The **destroy subject** operation removes the row and column for the destroyed subject from the access matrix. The **create object** and **destroy object** operations are much like their subject counterparts, except that they work on a column-only basis.

Two examples of TAM commands are given below.

command *create-file*$(U : user, F : file)$
  create object F **of type** $file$;
  enter *own* **in** $[U, F]$
end

command *transfer-ownership*$(U : user,$
  $V : user, F : file)$
  if $own \in [U, F]$ **then**
   delete *own* **from** $[U, F]$;
   enter *own* **in** $[V, F]$;
end

The first command authorizes users to create files, with the creator becoming the owner of the file. The second command allows ownership of a file to be transferred from one user to another.

2.2 Summary of TAM

To summarize, a system in specified in TAM by defining the following finite components.

1. A set of rights R.
2. A set of types T.
3. A set of state-changing commands, as defined above.
4. The initial state.

We say that the rights, types and commands define the system *scheme*. Note that once the system scheme is specified by the security administrator it remains fixed thereafter for the life of the system. The system state, however, changes with time.

2.3 The Augmented TAM (ATAM) Model

ATAM was defined in [2] to be TAM extended with ability to test for the absence of a right in a cell of the access matrix. In other words, a test of the form $r_i \notin [X_{s_i}, X_{o_i}]$ may be present in the condition part of ATAM commands. Ammann and Sandhu argue (informally) that dynamic separation of duties requires this ability to test for absence of access rights. They show how transaction control expressions [6] can be specified in ATAM.

A surprising result of this paper is that TAM and ATAM are formally equivalent in expressive power. However, as we will see, from a practical viewpoint our construction suggests that it is beneficial to allow testing for absence of access rights.

3 Equivalence of TAM and ATAM

In this section we give a construction to show the equivalence of TAM and ATAM. We first show in section 3.1 that ATAM schemes without **create** or **destroy** operations can be reduced to TAM schemes. We then show, in section 3.2, how ATAM schemes with just **create** and **destroy** commands can be simulated in TAM. Finally in section 3.3 we give a procedure which converts any given ATAM scheme into an equivalent TAM scheme.

3.1 Equivalence Without Create or Destroy Operations

We now prove the equivalence of TAM and ATAM in the absence of create and destroy operations. Recall that ATAM extends TAM by allowing commands to test for absence of access rights in the condition part. Thus, TAM is a restricted version of ATAM. To establish equivalence we therefore need to show that every ATAM system can be simulated by a TAM system. This is done by giving a procedure to construct a TAM system that can simulate a given ATAM system.

The basic idea in the construction is to represent the absence of rights in the ATAM system by the presence of *complementary rights* in the TAM system. Suppose that the given ATAM system has set of rights R. In the TAM simulation we include the rights R, as well as the complementary rights $\bar{R} = \{\bar{x} \mid x \in R\}$. The construction will ensure that

$$x \in [S_i, O_j] \Leftrightarrow \bar{x} \notin [S_i, O_j]$$

The initial state of the TAM access matrix has all the rights of the initial matrix of ATAM, as well as all the complementary rights implied by the above predicate.

If the ATAM system has no creation operations, the following procedure constructs an equivalent TAM system.

1. Whenever a right x is entered in a cell of the ATAM system, it is also entered in the identical cell in the TAM system; but, moreover, the complementary right $\bar{x}$ is deleted from that cell in the TAM system.

2. Similarly whenever a right x is deleted from a cell of the ATAM system, it is deleted from the identical cell in the TAM simulation. At the same time, the complementary right $\bar{x}$ is entered in that cell in the TAM simulation.

3. Also if an ATAM command tests for the absence of some rights, than the corresponding TAM command produced by our construction tests for the absence of rights by means of testing for presence of complementary rights. For example, the test $x \notin [S, O]$ in an ATAM command will be simulated by the test $\bar{x} \in [S, O]$ in the TAM system.

We have the following result.

Theorem 1 For every ATAM system S_1 the construction outlined above produces an equivalent TAM system S_2.

Proof Sketch: It can be easily seen that any reachable state in S_1 can be reached in S_2 (and vice-versa), if each ATAM command is simulated by the TAM command constructed as above (and vice versa). In particular the TAM system will maintain the invariant $x \in [S_i, O_j] \Leftrightarrow \bar{x} \notin [S_i, O_j]$ for all cells in the access matrix. □

3.2 Equivalence With Create and Destroy Operations

The occurrence of create operations in the given ATAM system considerably complicates the construction. We will focus only on creation of subjects, since every ATAM subject or object will be simulated in the TAM system as a subject (i.e., every column has a corresponding row in the access matrix). In other words the access matrix of the TAM system is square. This entails no loss of generality, since ATAM subjects are not necessarily active entities.

A primitive "**create subject** S_j" operation introduces a new empty row and column in the access matrix. To follow through with the complementary rights construction, we need to introduce the $\bar{R}$ rights in *every* cell involving S_j. The number of primitive operations required to do this is directly proportional to the number of subjects existing, at that moment, in the system. Since this is a variable number, a single TAM command cannot achieve this result. Instead we must use a sequence of TAM commands. The number of TAM commands required is unbounded, being directly proportional to the size of the access matrix.

3.2.1 Linked List Structure

To facilitate the TAM simulation, our construction organizes the subjects in a linked list structure, which can be traversed by following its pointers. The pointers, and the head and tail locations of the list, are easy to implement by rights in the access matrix. New subjects are inserted at the tail of the list when they are created. In order to fill up the row and column for the new subject with the complementary rights $\bar{R}$, the list is traversed from head to tail entering $\bar{R}$ in the cells for the new subject along the way.

Three new rights *head*, *tail*, and *next* are introduced in the initial state of the matrix. The right *head* in a cell $[S_i, S_i]$ of the matrix implies that S_i is the first subject in the linked list. Similarly, the right *tail* in a cell $[S_i, S_i]$ of the matrix implies that S_i is the last subject in the linked list. The right *next* in a cell $[S_i, S_j]$ of the matrix implies that S_j is the successor to S_i in the linked list (or equivalently that S_i is the predecessor to S_j in the list). In addition, two rights C and tr are used for book-keeping purposes in the simulation, as will be explained in section 3.2.2. It is assumed, without loss of generality, that rights *next, head, tail, token, C,* and *tr* are distinct from the rights in the given ATAM system.

A create operation in an ATAM system is simulated by multiple commands in the TAM system. The key to doing this successfully is to prevent other TAM commands from interfering with the simulation of the given ATAM command. The simplest way to do this is to ensure that ATAM commands can be executed in the TAM simulation only one at a time. To do this we need to synchronize the execution of successive ATAM commands in the TAM simulation. Thus the problem of simulating ATAM in TAM requires solution of a synchronization problem. Synchronization is achieved by introducing an extra subject called SNC, and an extra right *token* as shown in figure 1. The role of SNC is to sequentialize the execution of simulation of ATAM commands in the TAM system. The type of SNC is *snc*, which is assumed, without loss of generality, to be distinct from any type in the given ATAM system.

To summarize, the initial state of the TAM system consists of the initial state of the given ATAM system augmented in three respects.

1. First, an empty row is introduced for every ATAM object, which does not have a row in the given ATAM access matrix. The *head*, *tail* and *next* rights are introduced to order the subjects in a linked list.
2. Secondly, complementary rights are introduced as per the following predicate:
$$x \in [S_i, S_j] \Leftrightarrow \bar{x} \notin [S_i, S_j]$$
3. Thirdly, the SNC subject is introduced in the access matrix with $[SNC, SNC]$= *token*, and all other cells involving SNC being empty.

3.2.2 Simulation of ATAM create commands

We now consider how the ATAM command *CCreate* given below can be simulated by several TAM commands.

$$
\begin{array}{l}
\textbf{command } CCreate(S_1 : s_1, S_2 : s_2, \ldots, \\
\qquad\qquad\qquad\qquad\qquad S_m : s_m, S_c : s_c) \\
\quad \textbf{if } \alpha(S_1, S_2, \ldots, S_m) \textbf{ then} \\
\qquad \textbf{create subject } S_c \textbf{ of type } s_c \\
\textbf{end}
\end{array}
$$

The name *CCreate* is a mnemonic for conditional creation. This command tests for the condition $\alpha(S_1, S_2, \ldots, S_m)$. If the condition is true, the command creates a new subject S_c.

The TAM simulation of *CCreate* proceeds in three phases, respectively as illustrated in figures 2, 3 and 4. In these figures we show only the relevant portion of the access matrix, and only those rights introduced specifically for the TAM simulation. Complementary rights are shown only in the cells involving the newly created subject S_c. It is understood that the original ATAM rights are distributed exactly as in the ATAM system, along with complementary rights required to maintain the predicate $x \in [S_i, S_j] \Leftrightarrow \bar{x} \notin [S_i, S_j]$. In the figures, n represents the total number of subjects in the system prior to the create operation.

The first phase consists of a single TAM command *CCreate-I* which tests whether (i) the condition of the ATAM command $\alpha(S_1, S_2, \ldots, S_m)$ is true, and (ii)

	SNC	S_1	S_2	$\ldots$	S_n
SNC	*token*				
S_1		*head*	*next*		
S_2					
$\ldots$					
S_n					*tail*

Figure 1: Initial Access Matrix of the TAM Simulation

	SNC	S_1	S_2	$\ldots$	S_n	S_c
SNC						
S_1		*head*	*next*			
S_2						
$\ldots$						
S_n					*tail*	
S_c						*tr*

Figure 2: TAM Simulation of the ATAM Command $CCreate$: Phase I

whether $token \in [SNC, SNC]$. The former test is obviously required. The predicate α may involve tests for absence of access rights. Hence, in the TAM simulation we replace α by α', which is obtained by substituting tests for presence of complimentary rights in place of tests for absence of rights in α. The latter test for $token \in [SNC, SNC]$ ensures that the TAM simulation of $CCreate$ can commence only if no other ATAM command is currently being simulated. It also ensures that once phase I of the simulation of $CCreate$ has started, the simulation will proceed to completion before simulation of another ATAM command can begin. The phase I TAM command is given below.

command *CCreate-I*$(S_1 : s_1, S_2 : s_2, \ldots,$
$S_m : s_m, S_c : s_c, SNC : snc)$
  if $\alpha'(S_1, S_2, \ldots, S_m) \wedge$ $token \in [SNC, SNC]$
  then
    delete *token* **from** $[SNC, SNC]$;
    create S_c **of type** s_c;
    enter *tr* **in** $[S_c, S_c]$
end

The body of this command deletes *token* from $[SNC, SNC]$ and creates subject S_c. It also enters *tr* in $[S_c, S_c]$ indicating that all cells corresponding to S_c have to be traversed. The states of the access matrix, before and after execution of *CCreate-I*, are outlined in figures 1 and 2 respectively.

In phase II of the simulation the right C is passed, in turn, from $[S_c, S_1]$ to $[S_c, S_2]$ and so on to $[S_c, S_n]$. A right C in a cell of a matrix indicates that all complementary rights have to be introduced in that cell. Hence complementary rights $\bar{R}$ are introduced in the cell from which the right C is removed. The phase II commands are given below. The type of subjects S_i and S_r indicated in the commands by $T - snc$ implies that these subjects can be of any type in $T - snc$ (i.e., any type other than snc). Strictly speaking, we should have a separate command for each type in $T - snc$, but we allow this slight extension of our notation to simplify the presentation.

command *CCreate-1-II*$(S_c : s_c, S_i : T - snc)$
  if $tr \in [S_c, S_c] \wedge head \in [S_i, S_i]$ **then**
    delete *tr* **from** $[S_c, S_c]$;
    enter C **in** $[S_c, S_i]$
end

command *CCreate-2-II*$(S_c : s_c, S_i : T - snc,$
$S_r : T - snc)$
  if $next \in [S_i, S_r] \wedge C \in [S_c, S_i]$ **then**
    enter $\bar{R}$ **in** $[S_c, S_i]$;
    enter $\bar{R}$ **in** $[S_i, S_c]$;
    delete C **from** $[S_c, S_i]$;
    enter C **in** $[S_c, S_r]$
end

The command *CCreate-1-II* tests if phase I is completed by looking for right *tr*, which it then removes

	SNC	S_1	S_2	...	S_n	S_c
SNC						
S_1		$head$	$next$			
S_2						
...						
S_n					$tail$	
S_c		C				

(a) After execution of *CCreate-1-II*

	SNC	S_1	S_2	...	S_n	S_c
SNC						
S_1		$head$	$next$			$\bar{R}$
S_2						
...						
S_n					$tail$	
S_c		$\bar{R}$	C			

(b) After one execution of *CCreate-2-II*

	SNC	S_1	S_2	...	S_n	S_c
SNC						
S_1		$head$	$next$			$\bar{R}$
S_2						$\bar{R}$
...						$\bar{R}$
S_n					$tail$	
S_c		$\bar{R}$	$\bar{R}$	$\bar{R}$	C	

(c) End of phase II

Figure 3: TAM Simulation of the ATAM Command $CCreate$: Phase II

	SNC	S_1	S_2	...	S_n	S_c
SNC	$token$					
S_1		$head$	$next$			$\bar{R}$
S_2						$\bar{R}$
...						$\bar{R}$
S_n						$next, \bar{R}$
S_c		$\bar{R}$	$\bar{R}$	$\bar{R}$	$\bar{R}$	$tail, \bar{R}$

Figure 4: TAM Simulation of the ATAM Command $CCreate$: Phase III

and enters right C in the head column of the list of subjects. Command *CCreate-2-II* introduces complementary rights in all cells involving S_c except the tail subject by passing right C along the linked list of subjects. The insertion of complimentary rights in the tail column of the linked list is deferred until Phase III. The execution of Phase II commands is illustrated in figure 3.

In phase III of the simulation, the new subject S_c is inserted at the end of the linked list. At the same time, complementary rights are introduced in the previous and the new tail cells. Also *token* is introduced in the cell $[SNC, SNC]$ indicating that the simulation of *CCreate* is complete. The Phase III command is given below.

Command *CCreate-III*$(S_c : s_c, S_n : T - snc,$
$SNC : snc)$
if $C \in [S_c, S_n] \wedge tail \in [S_n, S_n]$ **then**
delete C **from** $[S_c, S_n]$;
enter $\bar{R}$ **in** $[S_c, S_n]$;
enter $\bar{R}$ **in** $[S_n, S_c]$;
enter $\bar{R}$ **in** $[S_c, S_c]$;
enter *next* **in** $[S_n, S_c]$;
delete *tail* **from** $[S_n, S_n]$;
enter *tail* **in** $[S_c, S_c]$;
enter *token* **in** $[SNC, SNC]$
end

Prior to execution of the *CCreate-III* command we have the situation shown in figure 3(c). After execution of *CCreate-III* we have the situation of figure 4. The TAM simulation is now ready to proceed with execution of another ATAM command.

3.2.3 Simulation of ATAM destroy commands

In order to simulate creation,we have seen that the subjects need to be related in a linked list structure. Hence, whenever a subject is destroyed the linked lists should still be maintained. For instance, when subject S_3 is destroyed in context of figure 5(a), we should maintain the linked list as shown in figure 5(b).

To be concrete, consider the following ATAM command whose name *CDestroy* is a mnemonic for conditional destroy.[1]

command *CDestroy* $(S_1 : s_1, S_2 : s_2, ..S_m : s_m,$
$S_d : s_d)$
if $\alpha(S_d, S_1, S_2, \ldots, S_m)$ **then**
destroy subject S_d
end

This command can be simulated by a single TAM command, since maintenance of the linked list requires adjustment to a fixed number of cells of the access matrix. However, we do need several variations of the TAM command, depending upon whether the subject being destroyed is in the middle, or at the head or tail of the linked list. We also need a variation to simulate the extreme case where the subject being destroyed is the only subject in the linked list.

The TAM command to simulate *CDestroy*, when S_d is in the middle of the list, is as follows.

command *CDestroy-middle* $(S_d : s_d, S_l : T{-}snc,$
$S_r : T - snc, S_1 : s_1, S_2 : s_2, \ldots,$
$S_m : s_m, SNC : snc)$
if $\alpha'(S_d, S_1, S_2, \ldots, S_m) \wedge$
$token \in [SNC, SNC] \wedge$
$next \in [S_l, S_d] \wedge next \in [S_d, S_r]$
then
destroy subject S_d;
enter *next* **in** $[S_l, S_r]$
end

As done in section 3.2.2, the predicate α' is obtained by substituting tests for presence of complimentary rights in place of tests for absence of rights in α. The test for the *token* right ensures that ATAM commands are simulated one at a time. The tests for *next* ensure that S_l and S_r are respectively the predecessor and successor of S_d in the linked list. The body of the command maintains the linked list.

If S_d is at the head or at the tail of the linked list, we respectively have the following two commands.

command *CDestroy-head* $(S_d : s_d, S_r : T - snc,$
$S_1 : s_1, S_2 : s_2, \ldots, S_m : s_m,\ SNC : snc)$
if $\alpha'(S_d, S_1, S_2, \ldots, S_m) \wedge$
$token \in [SNC, SNC] \wedge$
$head \in [S_d, S_d] \wedge next \in [S_d, S_r]$
then
destroy subject S_d;
enter *head* **in** $[S_r, S_r]$
end

command *CDestroy-tail* $(S_d : s_d, S_l : T - snc,$
$S_1 : s_1, S_2 : s_2, \ldots, S_m : s_m, SNC : snc)$
if $\alpha'(S_d, S_1, S_2, \ldots, S_m) \wedge$
$token \in [SNC, SNC] \wedge$
$next \in [S_l, S_d] \wedge tail \in [S_d, S_d]$

[1] We have shown S_d as occuring in the conditional predicate α. This should be interpreted as saying that S_d may optionally occur in the condition, but is not required to.

	SNC	S_1	S_2	S_3	S_4	S_5
SNC	*token*					
S_1		*head*	*next*			
S_2				*next*		
S_3					*next*	
S_4						*next*
S_5						*tail*

(a) Before destruction of S_3

	SNC	S_1	S_2	S_4	S_5
SNC	*token*				
S_1		*head*	*next*		
S_2				*next*	
S_4					*next*
S_5					*tail*

(b) After destruction of S_3

Figure 5: Destruction of S_3

> **then**
>   **destroy subject** S_d;
>   **enter** *tail* **in** $[S_l, S_l]$
> **end**

Finally, for the extreme case where S_d is the only subject in the linked list we have the following TAM command.

> **command** *CDestroy-last* $(S_d : s_d,\ SNC : snc)$
>   **if** $\alpha'(S_d) \wedge$ $token \in [SNC, SNC] \wedge$
> $head \in [S_d, S_d] \wedge tail \in [S_d, S_d]$
>   **then**
>   **destroy subject** S_d
> **end**

3.3 Simulation of ATAM schemes

So far we have seen how ATAM commands which do not have create and destroy operations, and ATAM commands which just have either create or destroy operations are converted into TAM commands. A general procedure for simulating an arbitrary ATAM command in TAM can be obtained by combining these ideas. Consider a ATAM command with multiple operations (i.e., a sequence of **enter**, **delete**, **create**, and **destroy** operations). Based on the previous discussion, we know how to simulate each primitive operation in turn. With some additional book-keeping we can keep track of a "program counter" which moves down the sequence of primitive operations in the body of the ATAM command, as each one gets simulated. The details are lengthy and tedious, and are omitted here.

We conclude this section by stating the central result of this paper.

Theorem 2 TAM and ATAM are formally equivalent in expressive power.

Proof Sketch: Follows from the above discussion. □

4 Conclusion

In this paper we have shown that the expressive power of the typed access matrix (TAM) model [7], and the augmented TAM (ATAM) model [2] are formally equivalent. Thus, testing for absence of access rights does not provide additional expressive power.

The TAM simulation of ATAM, given in this paper, clearly requires traversal of all subjects in the system whenever a create operation occurs in the given ATAM system. Morevoer, no other command can be initiated while this traversal is in progress. In real systems this will not be practically feasible.

The indication therefore is that ATAM can theoretically be reduced to TAM, but practically testing for absence of access rights appears to be useful. It is an open question whether this claim can be formalized, and proven on the basis of some formal complexity measure.

References

[1] Ammann, P.E. and Sandhu, R.S. "The Extended Schematic Protection Model." *Journal of Computer Security*, in press.

[2] Ammann, P.E. and Sandhu, R.S. "Implementing Transaction Control Expressions by Checking for Absence of Access Rights." *Proc. Eighth Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[3] Harrison, M.H., Ruzzo, W.L. and Ullman, J.D. "Protection in Operating Systems." *Communications of ACM* 19(8), 1976, pages 461-471.

[4] Lipton, R.J. and Snyder, L. "A Linear Time Algorithm for Deciding Subject Security." *Journal of ACM* 24(3):455-464 (1977).

[5] Sandhu, R.S. "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes." *Journal of ACM* 35(2), 1988, pages 404-432.

[6] Sandhu, R.S. "Transaction Control Expressions for Separation of Duties." *Proc. Fourth Aerospace Computer Security Applications Conference*, Orlando, Florida, December 1988, pages 282-286.

[7] Sandhu, R.S. "The Typed Access Matrix Model." *IEEE Symposium on Research in Security and Privacy*, Oakland, CA. 1992, pages 122-136.

[8] Sandhu, R.S. and Suri, G.S. "Implementation Considerations for the Typed Access Matrix Model in a Distributed Environment." *Proc. 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, October 1992, pages 221-235.

[9] Snyder, L. "Formal Models of Capability-Based Protection Systems." *IEEE Transactions on Computers* C-30(3):172-181 (1981).