Formal Model and Policy Specification of Usage Control

XINWEN ZHANG, FRANCESCO PARISI-PRESICCE, RAVI SANDHU
George Mason University
and
JAEHONG PARK
Eastern Michigan University

The recent usage control model (UCON) is a foundation for next generation access control models with distinguishing properties of decision continuity and attribute mutability. A usage control decision is determined by combining authorizations, obligations, and conditions, presented as UCON_{ABC} core models by Park and Sandhu. Based on these core aspects, we develop a formal model and logical specification of UCON with an extension of Lamport's temporal logic of actions (TLA). The building blocks of this model include: (1) a set of sequences of system states based on the attributes of subjects, objects, and the system, (2) authorization predicates based on subject and object attributes, (3) usage control actions to update attributes and accessing status of a usage process, (4) obligation actions, and (5) condition predicates based on system attributes. A usage control policy is defined as a set of temporal logic formulae that are satisfied as the system state changes. A fixed set of scheme rules is defined to specify general UCON policies with the properties of soundness and completeness. We show the flexibility and expressive capability of this formal model by specifying the core models of UCON and some applications.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms: Security

Additional Key Words and Phrases: access control, usage control, security policy, formal specification

This research was partially supported by the National Science Foundation under grant CCR-0310776.
A preliminary version of this paper appeared under the title "A Logical Specification for Usage Control" in the *Proceedings of 9th ACM Symposium on Access Control Models and Technologies*, Yorktown Heights, New York, USA, June 2-4, 2004.
Authors' address: X. Zhang, F. Parisi-Presicce, and R. Sandhu, Department of Information and Software Engineering, George Mason University, 4400 University Dr., MSN 4A4, Fairfax, VA 22030; email: {xzhang6, fparisip, sandhu}@gmu.edu. J. Park, School of Technology Studies, Eastern Michigan University, Ypsilanti, MI 48197; email: jae.park@emich.edu.

1. INTRODUCTION

Traditional access control models such as lattice-based access control (LBAC) [Bell and LaPadula 1975; Denning 1976; Sandhu 1993] and role-based access control (RBAC) [Sandhu et al. 1996; Ferraiolo et al. 2001] primarily consider static authorization decisions based on subjects' pre-assigned permissions on target objects. Access matrix models such as HRU [Harrison et al. 1976] and TAM [Sandhu 1992] use a matrix to distribute permissions with the discretion of individual subjects. In policy-based authorization management systems [Bertino et al. 2001; Damianou et al. 2001; Jajodia et al. 2001; Jajodia et al. 1997], a centralized reference monitor (or distributed reference monitor with centralized administration) checks a subject's permission when access is requested, and the request is granted according to system security policies at the time of the access request. Once a subject is granted a permission, there are no more security checks for continued access.

Developments in information technology, especially in electronic commerce applications, require additional features for access control. In recent information systems, the usage of a digital object can be not only an instantaneous access or activity, like read and write, but also temporal and transient, such as payment-based online reading, metered by reading time or chapters, or a downloadable music file that can only be played 10 times. So a subject's permission may decrease, expire, or be revoked along with the usage of the object.

As traditional identity-based and role-based access control models cannot satisfy these purposes, UCON is recently proposed to be the next generation access control model that extends traditional access control models in multiple aspects [Park and Sandhu 2004] and fits new security requirements. In UCON, an access may be an instantaneous action, but may also be a process lasting for some duration with several related and subsequent actions. Actions and events during an access process may result in changes to the system state, such as subject or object attributes, or in changes in the status of an access (e.g., revoke an access). Usage control can be enforced before or during an access process, or both. A usage decision in UCON is made by policies of authorizations, obligations, and conditions (also referred as UCON_{ABC} core models). Authorization decisions are determined by policies based on the attributes of subjects and objects, and rights. Obligations are actions that are required to be performed before or during an access process. Conditions are environment restrictions that are required to be valid before or during an access. An extreme example of UCON is the traditional access control models, in which the authorization decision is made instantly when an access request is generated, and there is no further check after that. More generally, usage control is a comprehensive model to represent the underlying principles of existing access control models, as well as the access control requirements in digital rights management (DRM), trust management, and other modern information systems.

The distinguishing properties of UCON beyond traditional access control models are the continuity of access decisions and the mutability of subject attributes and object attributes. In UCON, authorization decisions are not only checked and made before an access, but may be repeatedly checked during the access and may revoke the access if some policies are not satisfied, according to the changes of the subject or object attributes, or environmental conditions. Mutability is a new concept introduced by UCON, but its features can be found in traditional access control models and policies. For example, in a Chinese Wall policy, if a subject accesses an object in a conflict-of-interest set, then he/she cannot access any other conflicting objects in the future. That means, the potential object list that the

subject can access (we can consider this a subject attribute) has been changed as a side-effect of a previous access. This change, consequently, will restrict the next access of this subject. History-based access control policies can be expressed by UCON with this feature of attribute mutability. Also, mutability is useful to specify dynamic constraints in access control systems, such as separation of duty (SoD) policies, cardinality constraints, etc. Another prospective area is consumable access. Consumable access is becoming an important aspect in many applications, especially in DRM. For example in a pay-per-use DRM application with fixed credit of a subject, the available access time decreases with ongoing access.

Continuity and mutability in UCON introduce interactive and concurrent concepts into access control. An access results in the update of subject or object attributes as side-effects. These changes, in turn, may result in the change of other ongoing or future accesses by the same subject, or to the same object, or some access that is implicitly related. That means, an access may change not only its own state, but also the state of other accesses.

Park and Sandhu [Sandhu and Park 2003; Park and Sandhu 2004] presented the concept of mutability and continuity, and a conceptual model of UCON, which consists of several core sub-models including authorizations, obligations, and conditions. The main contribution of this paper is to formalize UCON model with a temporal logic, while in previous work the model is informal and conceptual. As UCON fundamentally extends the underlying mechanism from traditional access control models, and comprehensively captures the new features of recent proposed security systems, a formalized specification of the principles of UCON and its flexibility is necessary. With a logical specification, we provide a tool to precisely define policies for system designers and administrators. With a conceptual and informal model, the capability to define policy is limited. Also, a logical specification provides the precise meaning of the new features of UCON, such as the mutability of attributes and the continuity of usage control decisions. Finally, to analyze the general properties of UCON models such as expressive power and safety problem, we need a formalized model.

We use an extended form of Lamport's temporal logic of actions (TLA) [Lamport 1994] to build our logic model and formal specification. The basic components include predicates between subject, object, and system attributes, as well as actions performed by the system or subjects. A usage control policy is a logic formula built from these components that has to be satisfied by a UCON model.

The rest of this paper is organized as follows. Section 2 shows a motivating example of usage control, especially the new features of continuity and mutability. Section 3 gives a brief introduction of UCON. Section 4 introduces TLA briefly. Section 5 presents the details of our logic model. Section 6 presents the specification of the core UCON authorization models with our logic model. Section 7 and Section 8 introduce the logical specification of obligation core models and condition core models, respectively. In Section 9 we propose a set of rules to specify a general UCON model, and the completeness and soundness properties. Section 10 illustrates the flexibility and expressive power of our logical model. Some related work in access control with temporal aspects is reviewed in Section 11. Finally, we give our conclusions and present some ongoing and future work in Section 12.

2. MOTIVATING EXAMPLE

In this section we present an example motivating the new features of UCON. Traditional access control models and policies have difficulties, or lack the flexibility to specify policies in these scenarios. This example is originally from [Park and Sandhu 2004].

Consider a DRM application with limited number of simultaneous usages, where an object o can only be accessed and simultaneously used by a maximum of 10 users at a time. Each new access request must be granted and there is only one access generated from a single user at any time. If the number of users accessing the object is 10, then one existing user's ongoing access is revoked when a new request is generated. There are different policies to determine which user's ongoing access must be revoked. Among them,

(a) revocation by start time: the longest usage is revoked.

(b) revocation by idle time: the usage with the longest idle time is revoked.

(c) revocation by total usage time: the usage with the longest accumulating usage time is revoked.

For these three different policies[1], we need to define different attributes for subjects and objects.

(a) For each subject, we define the starting time as an attribute. The list of accessing subjects is defined as an object attribute, and each time a new access request is generated, this attribute is updated by adding the requesting subject. In UCON terminology, this is a pre-update. If the total accessing number is already 10, then the accessing subject with the earliest start time is revoked, and the new access is permitted. When an access is ended by a subject or revoked by the system, the subject is removed from the object's accessing list. This is called a post-update.

(b) An object has the same attribute as in (a). Each subject has two attributes: the status of the subject with a value $busy$ or $idle$, and the continuous idle time in a single usage process. In order to monitor the idle time, the system has to check the status and update the idle time during the entire ongoing access by means of ongoing-update. Similar to (a), there are pre-update, revoking access, and post-update actions. Revocation is performed with respect to the longest idle access when the total count of ongoing accessing subjects is larger than 10.

(c) Here again an object has the same attribute as in (a). Each subject has an attribute of accumulating usage time to record the total usage time of this subject on this object over the subject and object life. Similar to (a) and (b), there are pre-update, revoking access, and post-update actions. Revocation is performed with respect to the subject with the longest usage time access when the total count of ongoing accessing subjects is larger than 10. In addition, there is a post-update on the subject attribute after the usage (either ended by a subject or revoked by the system) by adding this usage time to the subject's historically accumulating accessing time.

In this example, an access is a process that interacts not only with a subject, but also with the system and other related processes which are accessing or trying to access the same object concurrently. An access decision is no longer a single function of (subject, object, right), but depends on the attributes of the subject and the object involved in the

[1] These policies require specification of a tie-breaking rule which we ignore for sake of simplicity.

access, and may change the attributes of these entities. On the other side, an access is not a simple action, but consists of a sequence of actions and events not only from a subject, but also from the system.

3. USAGE CONTROL

In this section we first briefly review the general ideas of UCON, then discuss attribute mutability in UCON. The details of the conceptual UCON model can be found in [Sandhu and Park 2003; Park and Sandhu 2004].

3.1 UCON Model

As depicted in Figure 1, a usage control system has six components: subjects and their attributes, objects and their attributes, rights, authorizations, obligations, and conditions[2]. The authorizations, obligations and conditions are components of usage control decisions. An authorization permits or denies an access based on the subject and/or the object attributes. Obligations are activities that have to be performed by subjects before or during an access. Conditions are system environment restrictions, which are not related to subject or object attributes.

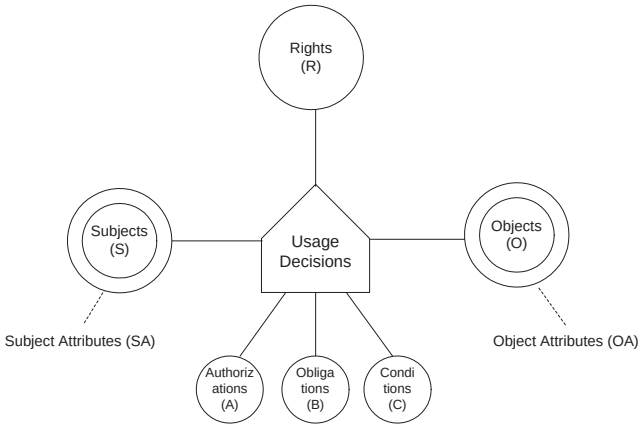

Fig. 1. Usage control model

The most important properties that distinguish UCON from traditional access control models and trust management are the continuity of usage decisions and the mutability of attributes. Continuity means that control decisions can be determined and enforced not only before an access, but also during the period of the access. Figure 2 shows a complete usage process consisting of three phases along the time line: before usage, ongoing usage, and after usage. The control decision components can be checked and enforced in the first two phases, named pre-decisions and ongoing-decisions respectively. In the after usage phase, no decision is checked and enforced since there is no access control after a subject finishes usage on an object. There can be obligations and conditions defined in this phase,

[2]Note that this diagram is slightly different from that in [Sandhu and Park 2003; Park and Sandhu 2004]. Here we place the usage decisions at the center instead of the rights.

which are called post-obligation and post-conditions, respectively. UCON is a session-based access control model, since it controls the current access request and ongoing usage. The obligations and conditions after an access are regarded as long-term obligations and conditions, which are not included in the core UCON models, but should be included in related administrative models. In this paper we only focus on the core aspects of UCON, while administrative models will be developed in the future.

Mutability means that subject and/or object attributes can be updated as the results of an access. Along with the three phases there are three kinds of updates: pre-updates, ongoing-updates, and post-updates. All these updates are performed and monitored by the system. An update of a subject or an object attribute in an access may result in a system action to allow or revoke current access or another access, according to the authorizations of the access. An update on the current usage may generate cascading updates, while an update on another access can act as an external event that would cause a change of the usage status, such as revocation. These are unique features of UCON models because of attribute mutability and decision continuity.

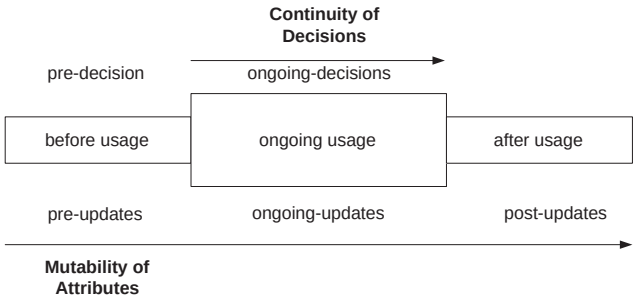

Fig. 2. Continuity and mutability properties of UCON

3.2 Attribute Management and Mutability

A usage control model is based on several underlying assumptions. In UCON, a usage decision is request-based, i.e., rights are not pre-assigned to subjects and permissions are computed at the time of usage requests. Authorization decisions are based on subject attributes and object attributes according to the usage control policies. Also, depending on the usage control policies, these attributes may have to be updated and their management is a key concern in usage control. Attribute management can be either "administrator-controlled" or "system-controlled".

Administrator-controlled attributes can be modified only by explicit administrative actions. These attributes are modified at the administrator's discretion but are "immutable" in that the system does not modify them automatically, unlike mutable attributes. Here the administrator can be either a security officer or a user, although in general, administrative actions are made by security officers. Administrator-controlled attributes are typical in traditional access control policies such as mandatory access control (MAC) and RBAC. Static separation of duty and user-role assignment in RBAC are other examples of this case.

Unlike administrator-controlled, in system-controlled attribute management, updates are the side effects or results of a subject's usage on objects. For instance, a subject's credit

balance is decreased by the value of the usage on an object at the time of the usage. This is different from the update by an administrative action because the update in this case is done by the system while in administrator-controlled management the update involves administrative decisions and actions. This is why system-controlled attributes are "mutable" attributes that do not require any administrative action for updates. Attribute mutability is considered as part of UCON core models. In this paper our concern lies in the system-controlled mutability issue, where updates are made as side effects of subjects' actions on objects. Five types of access control policies with system-controlled attribute mutability are summarized in [Park et al. 2004], including exclusiveness, accounting, immediate access revocation, obligations, and dynamic confinements.

4. TEMPORAL LOGIC OF ACTIONS

Extending temporal logic [Manna and Pnueli 1991] by introducing boolean valued actions, the temporal logic of actions (TLA) [Lamport 1994] is a powerful tool to specify systems and their properties, especially for interactive and concurrent systems. In this section we first give a brief introduction to the basic terms and the syntax of temporal formulae, and then introduce some additional temporal operators along with their semantics.

4.1 Building Blocks

Variables, values, and states are basic concepts in TLA. Values are elements of a data type. A variable has a name like x and y, and can be assigned a value. We assume that there is an infinite set of available variables with names x, y, etc., to which values can be assigned. A constant is a variable that is assigned with a fixed value. A state s is characterized by assignment of a value $s[\![x]\!]$ to each variable x.

A function is a non-boolean expression built from variables, operator symbols, and constants, such as $x^2 + y - 3$. The semantics $[\![f]\!]$ of a function f is a mapping from states to values. For example, $[\![x^2 + y - 3]\!]$ is the mapping that assigns to the state s the value $s[\![x]\!]^2 + s[\![y]\!] - 3$, where $s[\![x]\!]$ and $s[\![y]\!]$ denote the values that s assigns to x and y, respectively. Generally,

$$s[\![f]\!] \equiv f(\forall \text{`}v\text{'}\colon s[\![v]\!]/v)$$

where $f(\forall \text{`}v\text{'}\colon s[\![v]\!]/v)$ is the value obtained by substituting $s[\![v]\!]$ for each variable v in the expression. Formally, a variable is also a function that assigns the value $s[\![x]\!]$ to the state s.

A predicate is a boolean expression built from variables, operator symbols, and constants, such as $x = y + 1$. The semantics $[\![P]\!]$ of a predicate P is a mapping from states to boolean values. A state s satisfies a predicate P iff $s[\![P]\!]$, the value of $[\![P]\!]$ in s, equals $true$.

An action is a boolean-valued expression formed from variables, primed variables, operator symbols, and constants, such as $x' = y + 1$ and $x' - 1 \notin y'$. Formally, an action represents a relation between old states and new states, where unprimed variables refer to the old state and the primed variables refer to the new state. Formally, an action A is a function assigning a boolean $s[\![A]\!]t$ to a pair of states (s, t). For example, $x' = y + 1$ has the boolean value of $t[\![x]\!] = s[\![y]\!] + 1$. We say that (s, t) is an *A* step if $s[\![A]\!]t$ equals $true$. Generally,

$$s[\![A]\!]t \equiv A(\forall \text{`}v\text{'}\colon s[\![v]\!]/v, t[\![v]\!]/v')$$

Since a predicate P is a boolean expression built from variables and constants, it is regarded as a special action without primed variables. A pair (s, t) is a P step iff $s[\![P]\!]$ is $true$.

4.2 Temporal Formula and Semantics

The basic temporal operator is $\Box$ (*"Always"*). The semantics of a temporal action is defined using the concept of *behavior*. A behavior σ in TLA is an infinite sequence of states $< s_0, s_1, s_2, ... >$ (a finite set of states can be regarded as infinite with identical repeating states). With this idea, the semantics of an atomic formula with actions is defined as follows.

$$< s_0, s_1, s_2, ... > [\![A]\!] \equiv s_0[\![A]\!]s_1$$
$$< s_0, s_1, s_2, ... > [\![\Box A]\!] \equiv \forall n \geq 0 : s_n[\![A]\!]s_{n+1}$$

The same semantics can be defined for predicates since a predicate is a special form of action.

In TLA, a formula is built from predicates and actions with logical connectors and temporal operators. Recursively, a temporal formula is defined by the following grammar in BNF:

$$< formula >:\equiv < predicate > | < action > |\neg < formula > |$$
$$< formula > \wedge < formula > | < formula > \vee < formula > |$$
$$< formula > \rightarrow < formula > |\Box < formula > |$$

A formula is an assertion about a behavior. The semantic value $\sigma[\![F]\!]$ of a formula F is a boolean value on a behavior σ. Formally,

$$< s_0, s_1, s_2, ... > [\![F]\!] \equiv s_0[\![F]\!]s_1$$
$$< s_0, s_1, s_2, ... > [\![\Box F]\!] \equiv \forall n \geq 0 :< s_n, s_{n+1}, s_{n+2}, ... > [\![F]\!]$$

4.3 Extension of TLA

Other future operators, such as *"Eventually"* ($\Diamond$), can be defined using the "Always" ($\Box$) operator. The relationship between the "Always" and the "Eventually" operators can be expressed as:

$$\Diamond F \equiv \neg\Box\neg F$$

Based on the semantics of temporal actions and formulae, we can define other temporal operators and semantics similarly.

4.3.1 *The* "Next" *and* "Until" *Temporal Operator*. For a behavior $< s_0, s_1, s_2, ... >$, the semantics of the *Next* operator ($\bigcirc$) is defined as:

$$< s_0, s_1, s_2, ... > [\![\bigcirc F]\!] \equiv s_1[\![F]\!]s_2$$

Until ($\mathcal{U}$) is a binary operator. A formula $F\mathcal{U}G$ is $true$ if F is always $true$ until G is $true$ along the sequence of states. Formally,

$$< s_0, s_1, s_2, ... > [\![F\mathcal{U}G]\!] \equiv \exists i \geq 0 : \big(s_i[\![G]\!]s_{i+1} \wedge (0 \leq j < i \rightarrow s_j[\![F]\!]s_{j+1})\big)$$

Note that the semantics of $F\mathcal{U}G$ has no requirement on G for s_j and F for s_i and the following states, which is different from the "until" in the English language.

There is an equivalence between these temporal operators as the following shows.

$$\Diamond F \equiv (F \vee \neg F)\mathcal{U}F$$

4.3.2 *Past Temporal Operators.* TLA only defines future temporal operators like $\Box$ and $\Diamond$. In traditional temporal logic there are past temporal operators to specify the properties during the time preceding the current time. For a behavior $< s_0, s_1, s_2, ... >$ in TLA, if we consider s_0 as the state at the current time, then $s_1, s_2, ...$ are states of the future on the time line. We use the state sequence $..., s_{-2}, s_{-1}$ for states in the past along this time line. Therefore, a behavior is a state sequence:

$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... >$$

We can now define past temporal operators similar to the future ones: ■ (*Has-always-been*), ♦ (*Once*), $\ominus$ (*Previous*), $\mathcal{S}$ (*Since*). Formally,

$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![\blacksquare F]\!] \equiv \forall n < 0 : s_n[\![F]\!]s_{n+1}$$
$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![\blacklozenge F]\!] \equiv \exists n < 0 : s_n[\![F]\!]s_{n+1}$$
$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![\ominus F]\!] \equiv s_{-1}[\![F]\!]s_0$$
$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![F\mathcal{S}G]\!] \equiv$$
$$\exists i < 0 : \big(s_i[\![G]\!]s_{i+1} \wedge (i < j < 0 \rightarrow s_j[\![F]\!]s_{j+1})\big)$$

Similar to the future operators, there are some equivalences among these past operators, such as

$$\blacklozenge F \equiv \neg\blacksquare\neg F$$
$$\blacklozenge F \equiv F\mathcal{S}(F \vee \neg F)$$

5. LOGICAL MODEL OF UCON

In this section we present a logical approach for formalizing UCON. First we describe the basic components such as predicates and actions, then we define the logic model of UCON with these components.

5.1 Attributes and States

A system state is a set of assignments of values to variables. In UCON, there are three different kinds of variables: subject attributes, object attributes, and system attributes.

In UCON each entity (subject or object) is specified by a finite set of attributes. We require that each entity has at least one attribute for identity, called name, which is unique and cannot be changed. An attribute of an entity is denoted as $ent.att$ where ent is the subject or object's identity and att is the attribute name. Hereafter, we assume that an entity name without any attribute specified denotes its identity.

An attribute is a variable of a specific datatype, which includes a set of possible values (domain) and operators to manipulate them. In any state of the system, all attributes of an entity are assigned values from their corresponding domains. The datatype of an attribute depends on what kind of attribute it is, such as group membership, role, security clearance, credit amount, etc. The assignment of a value to an attribute is denoted by $ent.att = value$.

System attributes are variables that are not related to a subject or an object directly, such as system clock, location, etc. We define a special system attribute to specify the status of a single access process (s, o, r). Specifically, the function $state(s, o, r)$ is a mapping from $\{(s, o, r)\}$ to $\{initial, requesting, denied, accessing, revoked, end\}$. The semantics of the $initial$ state is that the access (s, o, r) has not been generated, while $requesting$ means the access has been generated and is waiting for the system's usage decision; $denied$ means

that the system has denied the access request according to the usage control policies before usage; $accessing$ means that the system has permitted the access and the subject has been accessing the object immediately after that. An access goes to the $revoked$ state when it is revoked by the system during the ongoing usage phase, or it goes to an end state when a subject finishes the usage.

In UCON, a function is an expression built from one or more attributes and constants. Formally, a function is a mapping from a set of attribute values to a new value. For instance, in the example in Section 2, the total number of ongoing accessing subjects for an object is a function of the object's attribute (a list of accessing subjects).

The variables for the attributes (including subjects, objects, and the system), the functions, and the constants comprise the basic terms of our logical model in UCON. A state of a UCON system is an assignment of values to all subject attributes, object attributes, and system attributes.

5.2 Predicates

A predicate is a boolean expression built from variables, functions, and constants, where variables includes subject attributes, object attributes, and system attributes. The semantics of a predicate is a mapping from system states to boolean values. A state satisfies a predicate if the attribute values assigned in this state satisfy the predicate. For example, the predicate $s.credit > \$100$ is $true$ if s's $credit$ attribute value in the current state of the system is larger than \$100. Since a system may have very different predicates from another system, the set of predicates for a general UCON model is not fixed. As examples, a unary predicate is built from one attribute variable and constants, e.g., $s.credit \geq \$100.00$, $o.classification = `supersecure'$. A binary predicate is built from two different attribute variables, e.g., $dominate(s.cleareance, o.classification)$, $s.credit \geq o.value$, $(s, r) \in o.acl$, where $o.acl$ is object o's access control list. Note that the two attributes in a binary predicate can be from a single subject or object, or one subject and one object, or from the system. A predicate can be defined with any number of attributes from a single entity, or two entities, or the system.

5.3 Actions

There are two types of actions in UCON: usage control actions and obligation actions.

5.3.1 *Usage Control Actions.* Usage control actions include actions to update attribute values and actions to change the status of a single access process ($state(s, o, r)$).

An update action changes the system state to a new state by updating the value of an attribute. Note that only subject and object attributes can be updated in UCON. How system attributes change is outside the scope of UCON model.

Corresponding to the point where an update is performed, there are three kinds of update actions defined in UCON: $preupdate$, $onupdate$, and $postupdate$. We distinguish these three types based on the phase where these actions are performed by the system: before usage, during usage, and after usage, respectively. Essentially, each of these actions updates an attribute value to a new value. In a real UCON system, an update action can have an arbitrary name specified by the system or policy designer. Also, a UCON model can have multiple updates in each phase for different attributes.

If the system performs an update successfully, the attribute value is changed to a new value, and the action is $true$, otherwise, it is $false$. Note that in our model we do not

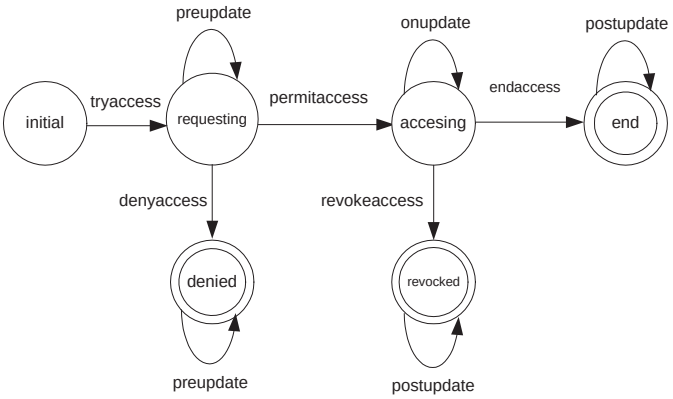

Fig. 3. State transition of a single access with usage control actions

consider the time delay of an action, and we assume that an action is always performed instantly causing the transition to the next state[3].

Another type of usage control action is performed by a subject or the system that change the status of an access (s, o, r). As mentioned before, there are six different possible values of $state(s, o, r)$ during an access life cycle. The transition from a state to another state is a usage control action, as shown in Figure 3. Note this figure only shows the changes of the usage status $state(s, o, r)$ in one usage process.

We can categorize all usage control actions into two classes: actions performed by a subject and actions performed by the system. Figure 4 shows these actions during the life cycle of a usage process. They are briefly explained below.

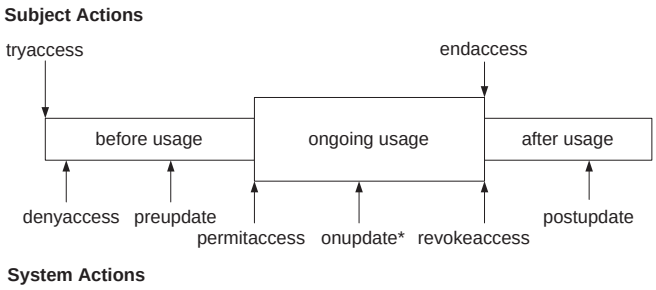

Fig. 4. Usage control actions

(1) $tryaccess(s, o, r)$: generates a new access request (s, o, r), performed by subject s.

[3]In a real system, an update may be delayed or failed, or an update is performed but the target object is not available, e.g., because of network problem or storage problem. Therefore there needs to be logging and recovering mechanisms to monitor the process. As standard approaches can be used for these purposes, the model does not capture these aspects for sake of simplicity.

(2) $permitaccess(s, o, r)$: grants the access request of (s, o, r), performed by the system.
(3) $denyaccess(s, o, r)$: rejects the access request of (s, o, r), performed by the system.
(4) $revokeaccess(s, o, r)$: revokes an ongoing access (s, o, r), performed by the system.
(5) $endaccess(s, o, r)$: ends an access (s, o, r), performed by subject s.
(6) $preupdate(attribute)$: updates a subject or an object attribute before granting access or after denying an access, performed by the system.
(7) $onupdate(attribute)$: updates a subject or an object attribute during the usage phase, performed by the system.
(8) $postupdate(attribute)$: updates a subject or an object attribute after access, performed by the system.

For a usage process, there can be multiple $preupdates$, $onupdate$, and $postupdate$ actions for different attributes before, during, and after the access, respectively. Also, in the ongoing usage phase there may be continual or periodical $onupdate$ actions for a single attribute, as the star symbol indicates in Figure 4.

5.3.2 *Obligation Actions.* In UCON an obligation is an action that must be performed by a subject before or during an access. For an access (s, o, r), an obligation is an action described by $ob(s_b, o_b)$, where ob is the obligation action name, and s_b and o_b are the obligation subject and object, respectively. Note that s_b and o_b may be the same as s and o, or different, depending on particular applications. For example, the downloading of a music file may require the requesting subject to click a privacy button. The obligation is defined as

$$click\_privacy(s, privacy\_statement)$$

where the obligation subject is the same as the accessing subject, and $privacy\_statement$ is the obligation object. As another example, a child's watching an online movie may need one of the parents' agreement in advance, where the obligation subject (parent) is different from the accessing subject. To identity what kind of obligations are required for a usage process, predicates can be defined based on the attributes of the requesting subject (s), the target object (o), the obligation subject (s_b), and the obligation object (o_b). For example, if a parents' obligation (e.g., clicking a statement) is needed before a child can watch online movies, then a predicate can be defined to specify the relationship between the access requesting subject (the child) and the obligation subject (the parent), and the obligation action for access $(s, o, watch)$ can be defined as

$$(s.parent = s_b) \wedge click(s_b, statement)$$

Note that in an obligation action, predicates are not used for control decisions, but for identifying what obligations are required. That is, an obligation action can always be performed whenever required, so that an obligation is not dependent on other permissions.

5.4 Model and Satisfaction of Formulae

With the predicates and actions that have been introduced, we can define a logical model of UCON.

Definition 5.1. A logical model of UCON is a 5-tuple: $\mathcal{M} = (\mathcal{S}, \mathcal{P}_{\mathcal{A}}, \mathcal{P}_{\mathcal{C}}, \mathcal{A}_{\mathcal{A}}, \mathcal{A}_{\mathcal{B}})$, where

—$\mathcal{S}$ is a set of sequences of system states,
—$\mathcal{P}_{\mathcal{A}}$ is a finite set of authorization predicates built from the attributes of subjects and objects,
—$\mathcal{P}_{\mathcal{C}}$ is a finite set of condition predicates built from the system attributes,
—$\mathcal{A}_{\mathcal{A}}$ is a finite set of usage control actions,
—$\mathcal{A}_{\mathcal{B}}$ is a finite set of obligation actions.

A state is a set of assignments of values to attributes, that is, a function on the set of subjects and their attributes, the set of objects and their attributes, and the set of system attributes. The set $\mathcal{A}_{\mathcal{A}}$ includes update actions and the actions changing the status of an access (s, o, r).

A preassumption of our logical model is that all predicates and actions are computable, e.g., a predicate is a computable function of attribute values. In practice they would need to be efficiently computable.

A logical formula is built from predicates and actions with logic connectors and temporal operators.

Definition 5.2. A logical formula in UCON is defined by the following grammar in BNF:

$$\phi ::= a|p|(\neg\phi)|(\phi \wedge \phi)|(\phi \rightarrow \phi)|\Box\phi|\Diamond\phi|\bigcirc\phi|\phi\mathcal{U}\phi|\blacksquare\phi|\blacklozenge\phi|\ominus\phi|\phi\mathcal{S}\phi|$$

where a is an action, p is a predicate.

If in a state sequence sq of a model $\mathcal{M}$, a state s satisfies a formula ϕ, we write $\mathcal{M}, sq, s \vDash \phi$. The satisfaction relation $\vDash$ is defined by induction on the structure of ϕ and only for $s_0 \in sq$. Specifically,

(1) $\mathcal{M}, sq, s_0 \vDash p$ iff $s_0[\![p]\!]$, where $p \in \mathcal{P}_{\mathcal{A}} \cup \mathcal{P}_{\mathcal{C}}$.
(2) $\mathcal{M}, sq, s_0 \vDash a$ iff $s_0[\![a]\!]s_1$, where $a \in \mathcal{A}_{\mathcal{A}} \cup \mathcal{A}_{\mathcal{B}}$, and s_1 is the next state of s_0 in sq.
(3) $\mathcal{M}, sq, s_0 \vDash \neg\phi$ iff $\mathcal{M}, sq, s_0 \nvDash \phi$.
(4) $\mathcal{M}, sq, s_0 \vDash \phi_1 \wedge \phi_2$ iff $\mathcal{M}, sq, s_0 \vDash \phi_1$ and $\mathcal{M}, sq, s_0 \vDash \phi_2$.
(5) $\mathcal{M}, sq, s_0 \vDash \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, sq, s_0 \nvDash \phi_1$ or $\mathcal{M}, sq, s_0 \vDash \phi_2$.
(6) $\mathcal{M}, sq, s_0 \vDash \Box\phi$ iff $\forall n \geq 0 : \mathcal{M}, sq, s_n \vDash \phi$.
(7) $\mathcal{M}, sq, s_0 \vDash \Diamond\phi$ iff $\exists n \geq 0 : \mathcal{M}, sq, s_n \vDash \phi$.
(8) $\mathcal{M}, sq, s_0 \vDash \bigcirc\phi$ iff $\mathcal{M}, sq, s_1 \vDash \phi$.
(9) $\mathcal{M}, sq, s_0 \vDash \phi_1\mathcal{U}\phi_2$ iff $\exists i \geq 0 : \mathcal{M}, sq, s_i \vDash \phi_2 \wedge (0 \leq j < i \rightarrow \mathcal{M}, sq, s_j \vDash \phi_1)$
(10) $\mathcal{M}, sq, s_0 \vDash \blacksquare\phi$ iff $\forall n < 0 : \mathcal{M}, sq, s_n \vDash \phi$.
(11) $\mathcal{M}, sq, s_0 \vDash \blacklozenge\phi$ iff $\exists n < 0 : \mathcal{M}, sq, s_n \vDash \phi$.
(12) $\mathcal{M}, sq, s_0 \vDash \ominus\phi$ iff $\mathcal{M}, sq, s_{-1} \vDash \phi$.
(13) $\mathcal{M}, sq, s_0 \vDash \phi_1\mathcal{S}\phi_2$ iff $\exists i < 0 : \mathcal{M}, sq, s_i \vDash \phi_2 \wedge (i < j \leq 0 \rightarrow \mathcal{M}, sq, s_j \vDash \phi_1)$

6. SPECIFICATION OF AUTHORIZATION CORE MODELS

For each decision component in UCON, a number of core models are defined based on the phase where updates are performed. In this section we first briefly introduce the core authorization models, then present their policy specifications. Obligation and condition models are illustrated in the next two sections, respectively.

In authorization core models, usage control decisions are dependent on subject and object attributes, which can be checked and determined in the first two phases of an access. Based on possible updates in all three phases, eight core authorization models can be defined as below.

—$preA_0$: a usage control decision is determined by authorizations before the usage, and there is no attribute update before, during, or after this usage.

—$preA_1$: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated before this usage.

—$preA_2$[4]: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated during this usage.

—$preA_3$: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated after this usage.

—onA_0: usage control is checked and the decision is determined by authorizations during the usage, and there is no attribute update before, during, or after this usage.

—onA_1: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated before this usage.

—onA_2: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated during this usage.

—onA_3: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated after this usage.

For ongoing authorization core models, continuous decision checks capture not only the attribute changes from the local ongoing usage process, but also other related usage processes. For example, a subject attribute change due to the system administrator's action may revoke an ongoing access to an object if any of the authorization predicates of this access is no longer valid.

6.1 The Model $preA_0$

As presented in [Sandhu and Park 2003; Park and Sandhu 2004], most traditional access control models can be expressed as $preA_0$ model, in which an authorization decision is determined by the system before the access happens, and there is no update for subject or object attributes. The usage control policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge ... \wedge p_i)\big)$

where p_1, ..., p_i are predicates built from subject and/or object attributes, which are pre-authorization predicates. The $permitaccess$ action grants the permission to s and starts the access. This policy states that a $permitaccess$ action implies that the authorization predicates must be true "before" the current system state. Note that in $preA_0$, there are no attribute updates before the $permitaccess$ action.

There are several assumptions made in the policy specification for this and all other core models in this chapter. First, a UCON policy is referred as a set of logical formulae for a single usage process (s, o, r), that is, we focus on the specification of system state changes during a single usage process of a core model, while the interactions between

[4] $preA_2$ is not a core model in [Park and Sandhu 2004]. For generality we include it here. The same holds for $preB_2$ in the next section.

concurrent usage processes are not captured by our policy specifications. Also in this chapter, we assume that before an access request is generated, the requesting subject and the target object exist in the system, i.e., creating and destroying subjects and objects are not specified in our logical model.

Another assumption is that the time line is bounded during the life time of a single usage process. That is, the $tryaccess$ is always the first action in a single usage process, all past temporal operators do not refer to any state before the $tryaccess$ in the local usage process, and all future operators do not refer to any state after the next $tryaccess$ of the same subject to the same object.

Negated predicates are not required explicitly, since we can always define a new predicate equivalent to a negated one. A disjunctive form of authorization predicates can also be specified by having one policy for each component, so that for an access permission (s, o, r), a system may have multiple policies for it. In a single usage process, at least one of them is satisfied by the model.

All authorization policies in UCON are defined for positive permissions (to enable permissions). For an access request, if there is no policy to enable the permission according to the attribute values, then the access is denied by default. This is sometimes called the closed system assumption, whereby no policy is specified to deny an access in a system. The same holds for obligation and condition core models.

The policy defined above states that the authorization predicates are checked when an access requested is generated, and there is no other check before the $permitaccess$ action. An alternative approach is to check the authorization predicates just before an access is granted, as the following formula states.

$$permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge (p_1 \wedge \ldots \wedge p_i)$$

In this formula, the predicates are required to be true just in the state of the $permitaccess$ action, ignoring how attributes can change after the $tryaccess$ action and before this state. Another alternative policy more restrictive than the above two policies is:

$$permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge \big((p_1 \wedge \ldots \wedge p_i) \mathcal{S} permitaccess(s, o, r)\big),$$

which states that the authorization predicates must be true from the $tryaccess$ action to the $permitaccess$ action. We use the original one as our $preA_0$ policy specification as it is the least restrictive one, and satisfies the *minimum requirement* of a $preA_0$ model. Another reason is that, in the $preA_1$ model (in next subsection), attribute updates are defined after $tryaccess$ action and before $permitaccess$ action, and after these updates, the authorization predicates may not be true in $preA_1$, so the authorization check is performed when the access requested is generated. As $preA_1$ dominates $preA_0$ in the family of UCON core models [Park and Sandhu 2004], we use the same approach for $preA_0$.

Example 1 In mandatory access control (MAC), each subject is assigned a security clearance, and each object is assigned a security classification. Both clearance and classification are labels in a lattice structure. A subject's clearance and an object's classification are compared to enforce security policies, such as the simple property and the star property. If the security clearance and classification are defined as attributes of subjects and objects, respectively, MAC as in Bell-LaPadula can be expressed in UCON with two $preA_0$ policies as shown below.

(1) $permitaccess(s, o, read) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, read) \wedge dominate(s.clearance, o.classification)\big)$

(2) $permitaccess(s, o, write) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, write) \wedge dominate(o.classification, s.clearance)\big)$

where $dominate$ is a binary predicate on a subject's clearance and an object's classification, and $dominate(x, y)$ is true iff x is a higher level label in the lattice than y. □

Example 2 Discretionary access control (DAC) model with access control list (ACL) can be expressed with a $preA_0$ policy. A subject attribute is its identity, and an object attribute is an access control list acl of pairs (id, r), where id is a subject's identity, and r is a right with which this subject can access this object.

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge ((s.id, r) \in o.acl)\big)$ □

Besides MAC and DAC, many other examples of $preA_0$ policies can be similarly specified. Section 10 shows some of them.

6.2 The Model $preA_1$

In $preA_1$, an authorization decision is checked before an access, and there are one or more update actions before the system grants the permission to the subject. The usage control policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge ... \wedge p_i)\big)$
(2) $permitaccess(s, o, r) \rightarrow \blacklozenge preupdate(attribute)$

where $attribute$ is either a subject or an object attribute. The first rule is the same as in $preA_0$. The second rule says that when a $permitaccess$ occurs, there is a $preupdate$ action that occurred before it. For multiple updates on different attributes, this rule is:

$$permitaccess(s, o, r) \rightarrow \blacklozenge preupdate_1(attribute_1) \wedge \blacklozenge preupdate_2(attribute_2) \wedge \ldots$$

For simplicity we only include one update action in all core models. Also, without loss of generality, we assume that in each logical formula there is at most one update for an attribute, as multiple updates on the same attribute have the same effect as the last one.

The two rules in this policy can be specified in a single rule as the following shows.

$permitaccess(s, o, r) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge ... \wedge p_i)\big) \wedge \blacklozenge preupdate(attribute)$

According to the assumption mentioned in Section 6.1, the time line is bounded during the local usage process, so in this policy the "Once" operator does not refer to any past state before $tryaccess$ in a single usage process. Therefore in $preA_1$, the $preupdate$ action is performed after the $tryaccess$, the authorization predicates are required to be $true$ before the $preupdate$, and there is no constraint after the update action.

Example 3 In a DRM pay-per-use application, a subject has a numerical valued attribute of $credit$, and an object has a numerical valued attribute of $value$. A $read$ access can be approved when a subject's $credit$ is more than an object's $value$. Before the access can start, an update to the subject's $credit$ is performed by the system by subtracting the object's $value$. The policy is:

(1) $permitaccess(Alice, ebook1, read) \rightarrow \blacklozenge\big(tryaccess(Alice, ebook1, read) \wedge$
$(Alice.credit \geq ebook1.value)\big) \wedge \blacklozenge preupdate(Alice.credit)$
$preupdate(Alice.credit) : Alice.credit' = Alice.credit - ebook1.value$

This rule specifies that whenever Alice's credit is more than the value of ebook1, she can get the reading permission, and the granting of the permission to Alice implies an update of her credit. The $preupdate$ results in a new value of Alice's credit by subtracting ebook1's value from the original credit. □

6.3 The Model $preA_2$

In $preA_2$, an authorization decision is checked and enforced before an access, and there are one or more update actions during the usage process. Although these updates cannot change the decision regarding the current ongoing usage, they may affect other ongoing or subsequent accesses from this subject or to this object. The policy for $preA_2$ is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge ... \wedge p_i)\big)$

(2) $permitaccess(s, o, r) \rightarrow \Diamond\big(onupdate(attribute) \wedge \Diamond endaccess(s, o, r)\big)$

The first rule is the same as that in $preA_0$. The second rule states that there is an ongoing update before the $endaccess$ action and after the $permitaccess$ action. In case when an update is necessary in each state during the ongoing-usage phase, this rule is expressed as

$$permitaccess(s, o, r) \rightarrow onupdate(attribute)\, \mathcal{U}\, endaccess(s, o, r)$$

This rule states that after the $permitaccess$ action, the attribute is updated in each state "until" the $endaccess$ action. Since a $permitaccess$ action changes the value of $state(s, o, r)$ to $accessing$, and $endaccess$ changes it to end, this policy is equivalent to the following.

$$\Box\big((state(s, o, r) = accessing) \rightarrow onupdate(attribute)\big)$$

In $preA_2$, since the authorization check is performed before the access, there is no revocation during the usage process in this and other pre-authorization models.

For a more general case when the ongoing update of an attribute is only needed when particular predicates are true, (e.g., a subject's idle time is updated only when the access status is $idle$), the policy is:

$$\Box\big((state(s, o, r) = accessing) \wedge p_{u1} \ldots \wedge p_{uj} \rightarrow onupdate(attribute)\big)$$

where $p_{u1}, \ldots, p_{uj}$ are predicates that trigger the update action when they are satisfied.

6.4 The Model $preA_3$

In $preA_3$, an authorization decision is checked before the access, and there are one or more update actions after the usage process. The usage control policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge ... \wedge p_i)\big)$

(2) $endaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$

The first rule is the same as in $preA_2$. The second rule says that a $postupdate$ action must be performed by the system after an access is ended by a subject. Similar to $preA_2$, no authorization is enforced after granting the access, so there is no revocation in this model.

Example 4 In a DRM membership-based application, a reader s has attributes $expense$ and $readingGroup$, and a book o has attributes $readingGroup$ and $readingCost$. A subject can read any book in his/her own reading group. The policy is:

(1) $permitaccess(s, o, read) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, read) \wedge (s.readingGroup = o.readingGroup)\big)$

(2) $endaccess(s, o, read) \rightarrow \Diamond postupdate(s.expense)$
$postupdate(s.expense) : s.expense' = s.expense + o.readingCost$

In this example, the authorization policy states that if both s and o belong to the same reading group, s can read the book and his/her expense is updated by adding the cost of this book after the access.

6.5 The Model onA_0

In the pre-authorization models, there is no security check after a system grants a permission. In onA_0, authorizations are enforced during an access period. The usage control policy is given below.

(1) $\Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$

In this model, ongoing authorization predicates (p_1, ..., p_i) have to be satisfied in any state during the access period (after the action $permitaccess$), otherwise the access is revoked by the system immediately.

This policy can also be specified as the following formula with *"Until"* operator.

$$permitaccess(s, o, r) \rightarrow$$
$$(p_1 \wedge ... \wedge p_i)\, \mathcal{U}\, \big(revokeaccess(s, o, r) \vee endaccess(s, o, r)\big)$$

which indicates that if a usage is permitted, the authorization predicates are true until this usage process is revoked by the system or ended by the subject. Since the $revokeaccess$ action changes $state(s, o, r)$ from $accessing$ to $revoked$, and $endaccess$ action changes $state(s, o, r)$ from $accessing$ to end, this formula is equivalent to the original one. Similarly we can use both approaches in other ongoing models (in this and next two sections).

Since we are specifying the core aspects of UCON, pre-authorization rules are not included in ongoing-authorization models, and for simplicity the $tryaccess$ action implied by the $permitaccess$ action is ignored. The same holds for other ongoing core models in this paper. In practice, an application may require a combination of several core models. We discuss this in Section 9.

Example 5 In an organization, a user Bob (with role $employee$) has a temporary position to conduct a short-term project with a certificate of $temp\_cert$. While Bob is accessing some sensitive information, his digital certificate ($temp\_cert$) for this project is being checked repeatedly. If his certificate (number) is in the Certification Revocation List (CRL) of the organization, his temporary role membership is revoked and he cannot access the information any more. The policy is:

(1) $\Box\big(\neg((Bob.role = employee) \wedge (Bob.temp\_cert \notin CRL)) \wedge (state(Bob, o, r) = accessing) \rightarrow revokeaccess(Bob, o, r)\big)$

6.6 The Model onA_1

In onA_1, the authorization decision is enforced during the usage process, and there are one or more update actions before a subject starts to access an object. The control policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(attribute)$

(2) $\Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$

The first rule implies a pre-update action before the $permitaccess$ action, which is similar to $preA_1$. But unlike in $preA_1$, the pre-decision based on authorization predicates is ignored in this rule since there is no authorization check before a subject starts to access an object in onA_1.

6.7 The Model onA_2

In onA_2, there are one or more update actions during a usage period. The control policy is:

(1) $\Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $permitaccess(s,o,r) \rightarrow$
$\Diamond\big(onupdate(attribute) \wedge \Diamond(endaccess(s,o,r) \vee revokeaccess(s,o,r))\big)$

Again, in the second rule, we only specify that there is only one update action during the ongoing-usage phase. In applications where an update is required in every ongoing state, the third rule is changed to:

$$permitaccess(s,o,r) \rightarrow$$
$$onupdate(attribute)\,\mathcal{U}\,(endaccess(s,o,r) \vee revokeaccess(s,o,r))$$

Similar to $preA_2$, this rule can be specified as:

$$\Box\big((state(s,o,r) = accessing) \rightarrow onupdate(attribute)\big)$$

or, more generally,

$$\Box\big((state(s,o,r) = accessing) \wedge p_{u1} \ldots \wedge p_{uj} \rightarrow onupdate(attribute)\big)$$

where $p_{u1}, \ldots, p_{uj}$ are predicates that require the update when they are satisfied.

6.8 The Model onA_3

In onA_3, update action is required after a usage process. The control policy is:

(1) $\Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $endaccess(s,o,r) \rightarrow \Diamond postupdate(attribute)$

(3) $revokeaccess(s,o,r) \rightarrow \Diamond postupdate(attribute)$

In many applications, the update after an access ended by a subject, is different from the one after an access is revoked by the system, as shown in the second and third rules. Here we simply use the same action name of $postupdate$, but they may change an attribute to different values, or update different attributes. For example, an ended access may update the total usage time of the subject, while a revoked access may update another attribute to record the time and reason of this revocation for auditing purposes. If the updates are the same for two cases, these two rules can be combined as in

$$endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow \Diamond postupdate(attribute)$$

Example 6 Consider the usage control policies for the example in Section 2. In this example, an object attribute is a set of accessing subjects $accessingS = \{s | state(s,o,r) = accessing\}$. We also define the system $clock$ as a system attribute. For the different policies we define different subject attributes.

(a) *Revocation by the earliest start time*

We define the starting time ($startTime$) as a subject attribute. The usage control policy can be specified as a combination of onA_1 and onA_3 as follows.

(1) $permitaccess(s, o, r) \rightarrow$
$\blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(o.accessingS) \wedge \blacklozenge preupdate(s.startTime)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$
$preupdate(s.startTime) : s.startTime' = sys.clock$

(2) $\Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s, o, r) = accessing) \wedge (s.startTime = Min_{startTime}(o.accessingS)) \rightarrow revokeaccess(s, o, r)\big)$

(3) $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow \Diamond postupdate(o.accessingS) \wedge \Diamond postupdate(s.startTime)$
$postUpdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$
$postUpdate(s.startTime) : s.startTime' = null$

where $|o.accessingS|$ is the number of accessing subjects with object o, and $Min_{startTime}(o.accessingS)$ is the earliest start time from $accessingS$. The first rule is a onA_1 rule specifying that whenever a subject tries to access the object, there must be two pre-updates before the subject starts to access, one updating $accessingS$ by adding this requesting subject, and another updating $s.startTime$ by assigning the current system clock. The second rule says that when the total number of accessing users is larger than 10, and the subject's $startTime$ is the earliest one, its access is revoked. The third rule specifies two post-updates needed when the access is ended or is revoked, one updating $accessingS$ by removing the subject, and another one updating $s.startTime$ by assigning the value $null$, which means the subject is not involved in an access. The post-updates are the same for both $endaccess$ and $revokeaccess$ actions in this system.

(b) *Revocation by the longest idle time*

We define two subject attributes: the status of the usage ($status$ with value $busy$ or $idle$) and the accumulative idle time in a single usage period ($idleTime$). The usage control policy is a combination of onA_1, onA_2, and onA_3 as follows.

(1) $permitaccess(s, o, r) \rightarrow$
$\blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(o.accessingS) \wedge \blacklozenge preupdate(s.idleTime)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$
$preupdate(s.idleTime) : s.idleTime' = 0$

(2) $\Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s, o, r) = accessing) \wedge (s.idleTime = Max_{idleTime}(o.accessingS)) \rightarrow revokeaccess(s, o, r)\big)$

(3) $\Box\big((state(s, o, r) = accessing) \wedge (s.status = idle) \rightarrow onupdate(s.idleTime)\big)$
$onupdate(s.idleTime) : s.idleTime' = s.idleTime + 1$

(4) $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow \Diamond postupdate(o.accessingS)$
$postupdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$

where $Max_{idleTime}(o.accessingS)$ is the largest $idleTime$ in the object's $accessingS$ attribute. Rules (1) and (4) are similar to (1) and (3) in (a), respectively, except that in rule (1), one pre-update action is to initialize the subject's $idleTime$. In rule (2), the revocation is determined by the $s.idleTime$. Rule (3) specifies the mutability of the subject attribute by saying that there must be a continuous update of $s.idleTime$ performed by the system whenever the status of the subject is $idle$.

(c) *Revocation by the longest total usage time*

We define the accumulating usage time $usageTime$ as a subject attribute. The control

policy is a combination of onA_1 and onA_3 as follows.

(1) $permitaccess(s, o, r) \rightarrow$
$\blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(o.accessingS)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$

(2) $\Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s, o, r) = accessing) \wedge (s.usageTime = Max_{usageTime}(o.accessingS)) \rightarrow revokeaccess(s, o, r)\big)$

(3) $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow$
$\Diamond postupdate(s.usageTime) \wedge \Diamond postupdate(o.accesingS)$
$postupdate(o.accesingS) : o.accessingS' = o.accessingS - \{s\}$
$postupdate(s.usageTime) : s.usageTimes' = s.usageTime + sys.periodT$

where $Max_{usageTime}(o.accessingS)$ is the largest $usageTime$ in $accessingS$. Rule (1) is the same as in the previous case except that there is only one pre-update action; rule (2) specifies that the revocation is determined by the total usage time of the subject. Rule (3) says that after each usage, there must be an update on $usageTime$ by adding this usage time to the old value. Here $sys.periodT$ is a system attribute to record this accessing's period. A system attribute may be defined and updated repeatedly along a usage process to record a single access's period. While the update of system attributes is not included in UCON core models, for simplicity we just use an attribute to conceptually illustrate the post-update action. Note that the revocation is determined by a subject's historically accumulating total usage time before this ongoing access. The time of an ongoing access is not considered in the $usageTime$ attribute of a subject. □

7. SPECIFICATION OF OBLIGATION CORE MODELS

Obligations and conditions are two important components in the usage decision of UCON, besides authorizations. In this section we discuss the logical approach to obligations. The specification of conditions is discussed in the next section.

Because of the continuity of a usage decision, there are two types of obligations in UCON.

1. pre-obligations: obligations that must have been performed before a subject starts to access an object.
2. ongoing-obligations: obligations that must be performed during a usage process.

Obligations that have to be performed after an access, since they only affect the future usage process, are considered as global obligations [Sandhu and Park 2003; Park and Sandhu 2004]. For example, an action of a user clicking an agreement button before playing a music file is regarded as an obligation, while the payment action of a monthly billing is a global obligation, because this action does not affect the current usage access. In UCON an administration model is needed to capture global obligations. In this paper, we only focus on the session-based usage control model, in which only obligations before and during the usage process are considered. The global obligations will be described in our future work.

Similar to authorization core models, we distinguish different obligation core models based on the phase where updates are performed as shown below.

—$preB_0$: a usage control decision is determined by obligations before the access, and there is no attribute update before, during, or after the usage.

—$preB_1$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated before the usage.

—$preB_2$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated during the usage.

—$preB_3$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated after the usage.

—onB_0: usage control is checked and the decision is determined by obligations during the access, and there is no attribute update before, during, or after the usage.

—onB_1: usage control is checked and the decision is determined by obligations during the access, and one or more subject or object attributes are updated before the usage.

—onB_2: usage control is checked and the decision is determined by obligations during the access, and one or more subject or object attributes are updated during the usage.

—onB_3: usage control is checked and the decision is determined by obligations during the access, and one or more subject or object attributes are updated after the usage.

In ongoing obligation core models, obligation actions may be required continually (i.e., in each ongoing state of the system), like the satisfaction of predicates in ongoing authorization models. Ongoing obligation actions may also be needed periodically, or in any state when some conditions are satisfied, e.g., when an event happens. For example, a user has to click an advertisement at 30 minute intervals or every 20 web pages accessed. For these purposes, attribute predicates can be defined to specify the conditions when obligation actions are needed.

7.1 The model $preB_0$

Similar to the model $preA_0$, the policy of $preB_0$ is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge (\blacklozenge ob_1 \wedge \blacklozenge ob_2 \wedge \ldots \wedge \blacklozenge ob_i)$

where $ob_1, \ldots, ob_i$ are obligation actions for access (s, o, r). This rule requires that an access can be granted only after all the obligations are satisfied. The difference between $preB_0$ and $preA_0$ is that, in $preB_0$, an obligation is satisfied before an access requested is granted, and generally may not be performed in the same state, so that the "Once" operator is applied for each of them in the policy formula. In $preA_0$, instead, the authorization predicates are checked in a single state. As mentioned in Section 6.2, the "Once" operator does not refer to any state before the $tryaccess$ action in a single access process. This indicates that all obligation actions are for the current access request.

Note that here we just ignore the authorization factors (attribute predicates), since we are focusing on the obligation core model.

Example 7 In an online electronic marketing system, in order to place an order, a customer has to click a button to agree to the order policies. We define an action $click\_agreement$ as an obligation for each order, where the obligation subject is the same as the ordering subject, and the $agree\_statement$ is the obligation object. The usage control policy is:

(1) $permitaccess(s, o, order) \rightarrow$
$\blacklozenge tryaccess(s, o, order) \wedge \blacklozenge click\_agreement(s, agree\_statement)$

7.2 The Model $preB_1$

In $preB_1$, usage control is decided by obligations before the access, and there must be update(s) before the access. Similar to $preA_1$, the policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge (\blacklozenge ob_1 \wedge \blacklozenge ob_2 \wedge \ldots \wedge \blacklozenge ob_i) \wedge \blacklozenge preupdate(attribute)$

This rule is similar to that in $preB_0$ except that an update action must be performed after $tryaccess$ and before $premitaccess$, as the "Once" operator does not refer to any state before the $tryaccesss$ action in a single usage process.

7.3 The Model $preB_2$

Similar to $preA_2$, in $preB_2$ the usage control decision is checked before an access and update action(s) can be performed during the access. The policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge (\blacklozenge ob_1 \wedge \blacklozenge ob_2 \wedge \ldots \wedge \blacklozenge ob_i)$

(2) $permitaccess(s, o, r) \rightarrow \Diamond\big(onupdate(attribute) \wedge \Diamond endaccess(s, o, r)\big)$

For the case where an update is required in every state during the ongoing usage phase, the second rule becomes:

$$permitaccess(s, o, r) \rightarrow onupdate(attribute)\, \mathcal{U}\, endaccess(s, o, r)$$

or

$$\Box\big((state(s, o, r) = accessing) \rightarrow onupdate(attribute)\big)$$

or, more generally,

$$\Box\big((state(s, o, r) = accessing) \wedge p_{u1} \ldots \wedge p_{uj} \rightarrow onupdate(attribute)\big)$$

where $p_{u1}, \ldots, p_{uj}$ are predicates that require the update when they are satisfied.

7.4 The Model $preB_3$

Similar to $preA_3$, in $preB_3$ the obligations are checked before the access, and there are one or more update actions after the usage process. The usage control policy is:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge tryaccess(s, o, r) \wedge (\blacklozenge ob_1 \wedge \blacklozenge ob_2 \wedge \ldots \wedge \blacklozenge ob_i)$

(2) $endaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$

The first rule is the same as those in $preB_2$. The second rule says that a $postupdate$ action must be performed by the system after an access is ended by a subject. Since the control policy is not enforced after granting the access, there is no revocation in this and other pre-obligation models.

Example 8 In the Example in Section 7.1, a customer's $orderList$ is updated by adding the ordered item after he/she places an order. This can be expressed with a $preB_3$ policy as the following.

(1) $permitaccess(s, o, order) \rightarrow$
$\blacklozenge tryaccess(s, o, order) \wedge \blacklozenge click\_agreement(s, agree\_statement)$

(2) $endaccess(s, o, order) \rightarrow \Diamond postupdate(s.orderList)$
$postupdate(s.orderList) : s.orderList' = s.orderList \cup \{o\}$

7.5 The Model onB_0

In onB_0, the usage control policy is enforced during an access period. The policy is:

(1) $\Box\big(\neg(\bigwedge_i(p_{i1} \wedge \ldots \wedge p_{ik_i} \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

In this policy, ob_i is an obligation action required in an ongoing state of the system when predicates $p_{i1}, \ldots, p_{ik_i}$, defined on subject and/or object attributes, are true. Similar to onA_0, the policy specifies that after the $permitaccess$, either all the obligations are satisfied when the subject is $accessing$ the object, or the access is revoked immediately.

When obligations are required in every ongoing state, this policy is:

$$\Box\big(\neg(\bigwedge_i(true \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

or

$$\Box\big(\neg(\bigwedge_i ob_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

Example 9 In order to use an online provider service, an advertisement banner must be opened on the client's side, or the service is disconnected. This can be expressed in the onB_0 model as follows.

(1) $\Box\big(\neg open\_ad(s, ad\_banner) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

In this policy, $open\_ad$ is an obligation action on the obligation object ad_banner, that must be true during the whole accessing process.

7.6 The Model onB_1

In onB_1, there are one or more update actions before a subject starts to access an object. The policy is:

(1) $\Box\big(\neg(\bigwedge_i(p_{i1} \wedge \ldots \wedge p_{ik_i} \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $permitaccess(s,o,r) \rightarrow \blacklozenge tryaccess(s,o,r) \wedge \blacklozenge preupdate(attribute)$

The first rule is the same as in onB_0, while the second rule specifies that there is an update action before accessing the object. Since there is no usage control check before a subject starts to access an object, the second rule does not imply any obligation before the $permitaccess$ action.

7.7 The Model onB_2

In onB_2, there are one or more update actions during an access process. The policy is:

(1) $\Box\big(\neg(\bigwedge_i(p_{i1} \wedge \ldots \wedge p_{ik_i} \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $permitaccess(s,o,r) \rightarrow \Diamond\big(onupdate(attribute) \wedge \Diamond endaccess(s,o,r)\big)$

Similar to $preB_2$, for the cases where an update is required in every state during the ongoing access, the second rule becomes

$$permitaccess(s,o,r) \rightarrow onupdate(attribute)\, \mathcal{U}\, endaccess(s,o,r)$$

or

$$\Box\big((state(s,o,r) = accessing) \rightarrow onupdate(attribute)\big)$$

or, more generally,

$$\Box\big((state(s,o,r) = accessing) \wedge p_{u1} \ldots \wedge p_{uj} \rightarrow onupdate(attribute)\big)$$

where $p_{u1}, \ldots, p_{uj}$ are predicates that require the update when they are satisfied.

7.8 The Model onB_3

In onB_3, there must be update action(s) after a usage process. The control policy is:

(1) $\Box\big(\neg(\bigwedge_i(p_{i1} \wedge \ldots \wedge p_{ik_i} \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $endaccess(s,o,r) \rightarrow \Diamond postupdate(attribute)$

(3) $revokeaccess(s,o,r) \rightarrow \Diamond postupdate(attribute)$

Similar to onA_3, the post-update after an access is ended by a subject may be different from the one after an access is revoked by the system, as shown by different rules.

Example 10 In an online accessing application, a user needs to click an advertisement every 30 minutes. A subject attribute $UsageTime$ is the ongoing usage time in a single session. The policy can be specified as a combination policy of onB_1, onB_2, and onB_3 as follows.

(1) $\Box\big(\neg((s.UsageTime \ mod \ 30 = 0) \rightarrow click\_ad(s, ad\_banner)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$

(2) $permitaccess(s,o,r) \rightarrow \blacklozenge preupdate(s.UsageTime)$
$preupdate(s.UsageTime) : s.UsageTime' = 0$

(3) $\Box\big((state(s,o,r) = accessing) \rightarrow onupdate(s.UsageTime)\big)$
$onupdate(s.UsagtTime) : s.UsageTime' = s.UsageTime + 1$

(4) $endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow \Diamond postupdate(s.UsageTime)$
$postupdate(s.UsageTime) : s.UsageTime' = 0$

In this policy, the $click\_ad$ is an ongoing obligation action that must be performed when the $UsageTime$ is a multiple number of 30. Here $preupdate$ and $postupdate$ actions are needed to reset this attribute when the subject starts and ends (or be revoked by the system) the access, respectively. An ongoing update is used to record the accumulative usage time. Here we simplify this update by the increment of $UsageTime$ in each ongoing state.

8. SPECIFICATION OF CONDITION CORE MODELS

Conditions are environmental restrictions that have to be valid before or during a usage process. Formally, a condition is a state predicate built from system attribute(s). For example, a subject obtains a permission only when the system clock is in daytime, or in a particular period during daytime.

Based on the point when a condition for a usage is checked, there are two types of conditions:

1. pre-conditions: conditions that must be true before an access.
2. ongoing-conditions: conditions that must be true during the process of accessing an object.

Similar to the authorization and obligation core models, a set of core conditions models can be defined, by replacing the authorization predicates or obligation actions with system attributes in decision rules. For simplicity only the $preC_0$ and onC_0 core models are illustrated here. Note that in a condition core model, while the system attributes determine a usage decision, the system attribute changes are not captured in the model. As in authorization and obligation core models, all updates in a condition core model are performed on subject and/or object attributes.

The policy for the model $preC_0$ is expressed by:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (pc_1 \wedge ... \wedge pc_i)\big)$

where pc_1, ..., pc_i are condition predicates built from system attributes. This policy is very similar to that of $preA_0$ and $preB_0$, except that the decision is determined by predicates of system attributes, instead of the subject's and object's attributes in $preA_0$, and obligation actions in $preB_0$.

The policy of onC_0 is:

(1) $\Box\big(\neg(pc_1 \wedge ... \wedge pc_i) \wedge (state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$

This policy is similar to that of onA_0 and onB_0 except for the condition predicates.

Example 11 Suppose that a day-shift user (with role $dayshifter$) can access an object only during daytime. We define the local time $currentT$ as a system attribute, denoting an environment status, not an attribute of any subject or object. This is a combined model of $preA_0$, $preC_0$, and onC_0. The policy can be expressed as the following:

(1) $permitaccess(s, o, r) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, r) \wedge (s.role = dayshifter) \wedge (8am \leq currentT \leq 5pm)\big)$

(2) $\Box\big(\neg(8am \leq currentT \leq 5pm) \wedge (state(s, o, r) = accessing) \rightarrow$
$revokeaccess(s, o, r)\big)$

The first rule specifies the pre-authorization and pre-condition built from the subject's role name and the system time. The second rule specifies the ongoing condition built from the system time.

9. FORMAL SPECIFICATION OF GENERAL UCON MODELS

After specifying the core models in UCON, we study the formal semantics of a general UCON model in this section. Specifically, we show that a general UCON policy can be expressed with a set of logical formulae instantiated from a fixed set of scheme rules, and a set of logical formulae instantiated from these rules can be satisfied by at least one UCON model. These two properties are regarded as the completeness and soundness of our policy specification language.

9.1 Scheme Rules

In general, a usage control decision is determined by authorizations, obligations, and conditions. As shown in the core models in previous sections, authorizations are specified by predicates on subject and object attributes, obligations by subject actions, and conditions by predicates on system attributes. Therefore a general usage decision is a combination of these components.

For an access (s, o, r), let $pa_1, \ldots, pa_i$ be a set of authorization predicates, $ob_1, \ldots, ob_j$ be a set of obligation actions, and $pc_1, \ldots, pc_k$ be a set of condition predicates. According

to the specifications of the core models explained in previous sections, a UCON policy can be specified by two kinds of logical rules: a usage control decision rule and an update rule. The following control rules (CRs) are specified for the pre-decision and ongoing decision of a single usage process, respectively.

$CR1$: $permitaccess(s, o, r) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, r) \wedge (\bigwedge_{n_i} pa_{n_i}) \wedge (\bigwedge_{n_k} pc_{n_k})\big) \wedge (\bigwedge_{n_j} \blacklozenge ob_{n_j})$

$CR2$: $\Box\big(\neg\big((\bigwedge_{n_i} pa_{n_i}) \wedge (\bigwedge_{n_j}(pb_{n_j 1} \wedge \ldots \wedge pb_{n_j k_{n_j}} \rightarrow ob_{n_j})) \wedge (\bigwedge_{n_k} pc_{n_k})\big) \wedge$
$(state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$

where $1 \leq n_i \leq i$, $1 \leq n_j \leq j$, $1 \leq n_k \leq k$, and $pb_{n_j 1}, \ldots, pb_{n_j j k_{n_j}}$ are predicates to determine when the ongoing obligation ob_{n_j} is required.

An access request can be granted if its pre-decision components are true; while an ongoing access can be continued if all ongoing decision components are true. For an access, its pre-decision and ongoing decision components may or may not be the same.

The three types of update actions can be specified as the following update rules (URs).

$UR1$: $permitaccess(s, o, r) \rightarrow \blacklozenge preupdate(attribute)$

$UR2$: $permitaccess(s, o, r) \rightarrow \Diamond\big(ondupate(attribute) \wedge$
$\Diamond(endaccess(s, o, r) \vee revokeaccess(s, o, r))\big)$

$UR3$: $\Box\big((state(s, o, r) = accessing) \rightarrow onupdate(attribute)\big)$

$UR4$: $\Box\big((state(s, o, r) = accessing) \wedge p_{u1} \wedge \ldots p_{uj} \rightarrow onupdate(attribute)\big)$

$UR5$: $endaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$

$UR6$: $revokeaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$

where $UR1$ is for pre-updates, $UR2$, $UR3$, and $UR4$ are for ongoing updates, and $UR5$ and $UR6$ are for post-updates. Here $p_{u1}, \ldots, p_{uj}$ are predicates that trigger an update when satisfied during an access. For simplicity, we only include a single attribute in each update. Different rules can update the same attribute, or more generally different attributes. Also, a rule can update multiple attributes as we have explained in previous sections.

Both the control rules and update rules presented here are *schema* of real logical formulae in a UCON policy. A rule in a real system is an instantiations of one of these rules. A policy in the core models in previous sections can be specified by an instance formula of a control rule and an instantiated formula of an update rule. In general, a UCON policy can be a combination of multiple core models, which are specified by a set of the control rules and update rules.

9.2 Completeness and Soundness

The fixed set of scheme rules have the properties of completeness and soundness for UCON policy specification. Specifically, a UCON policy consists of a set of logical formulae, but at most one of them is instantiated from a scheme rule. For example, there is at most one formula instantiated from $CR1$, as any two of them can be combined into one with conjunctive authorization predicates, obligation actions, and condition predicates from both of them. On the other hand, for a set of logical formulae, each instantiated from a unique scheme rule, there is at least one model that can satisfy them.

THEOREM 9.1. *(Completeness) Any UCON policy can be specified by a non-empty set of control rules and a set of update rules, each of which is instantiated from a unique scheme rule.*

Proof. This is trivially true by definition, as from the construction of the control rules and update rules, we know $CR1$ and $CR2$ are not in conflict since they imply control decisions in different phases in a single usage process. The same holds for the update rules. Furthermore, the set of control rules specifies all possible decisions in a single usage process, and the set of update rules specify all possible updates in a single usage process. Therefore a general UCON policy can be specified by a non-empty set of control rules and a set of update rules. □

By completeness we mean that a general UCON model conceptually defined in [Park and Sandhu 2004] can be formally specified with our logical model. That is, the set of scheme rules is adequate to specify policies for all UCON core models and any combination of them.

THEOREM 9.2. *(Soundness) For a non-empty set of control rules and a set of update rules, each of which is instantiated from a unique scheme rule, there is at least one UCON model in which the system state transitions satisfy these rules.*

Proof. We construct a UCON model to satisfy eight logical formulae for a single access (s, o, r), one for a unique scheme rule. Consider the two control rules $CR1'$ and $CR2'$, which are instantiations of $CR1$ and $CR2$, respectively, and six update rules, $UR1'$, ..., $UR6'$, one for each unique scheme update rule, respectively. Without loss of generality, we assume that all the attributes in these update rules are different, since, as we have mentioned in Section 6, multiple updates on the same attribute can be reduced to a single update. Consider a system where a state is specified by the attributes (subject's, object's, and the system's) in all of the rules. Initially the system state is s_0, and $state(s, o, r) = initial$. The state transitions are constructed with the following steps and illustrated in Figure 5.

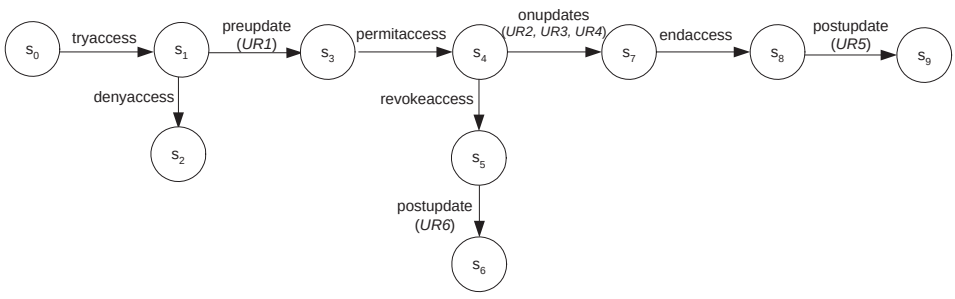

Fig. 5. State transitions

—In s_0, the subject s generates an access request ($tryaccess$) to o with right r, the value of $state(s, o, r)$ is changed to $requesting$, and the system's new state is s_1. The other attributes have the same values as in s_0.

—With the subject and object attributes and system attributes in s_1, if any of the predicates specified in $CR1'$ is not satisfied, or at least one obligation actions in $CR1'$ is not performed, then the system state changes via the action $denyaccess(s, o, r)$ to s_2, where $state(s, o, r) = denied$.

—In s_1, if all the predicates in $CR1'$ are satisfied, and all the obligations are performed by the corresponding subjects defined in $CR1'$, the update action in $UR1'$ is performed, and the system state changes to s_3.

—In s_3 the $permitaccess(s, o, r)$ action is performed by the system and the system state changes to s_4, where $state(s, o, r) = accessing$.

—If any predicate or obligation action included in $CR2'$ is not satisfied in s_4, the access is revoked, and system state changes to s_5, where $state(s, o, r) = revoked$.

—In s_5, the update action in $UR6'$ is performed, and the system state changes to s_6.

—If all the predicates and obligation actions included in $CR2'$ are satisfied in s_4, the update actions in $UR2'$ and $UR3'$ are performed by the system in s_4. If all the predicates in $UR4'$ are satisfied in s_4, perform the update action in $UR4'$ and the system state changes to s_7.

—In s_7 the subject s ends the access and the system state changes to s_8, where the system attribute $state(s, o, r) = end$.

—The update action in $UR5'$ is performed in s_8, and the system state changes to s_9.

With simple model checking, we can verify that all the rules are satisfied in these state transitions. That is, this model satisfies the set of logical formula. Therefore, any set of control rules and update rules can be satisfied by at least one UCON model. □

10. EXPRESSIVITY AND FLEXIBILITY

UCON is the first model to bring authorization, obligation, and condition together into access control. Both mutability and continuity are rarely discussed in traditional access control models and applications. In this section we apply the proposed logical specification language to show how to express policies in various applications.

10.1 Role-based Access Control Models

In RBAC [Sandhu et al. 1996], a role is a collection of permissions, and a permission is a pair (object, right) implying the right to the object. A role can be assigned to a user by an administrator or a security officer. A user can be assigned to a set of roles. In a session, a user activates a subset of his roles and obtains all the permissions associated with these activated roles. Roles may be organized in a partial order hierarchy, in which high-level roles (senior roles) inherit the permissions assigned to low-level roles (junior roles). RBAC can be expressed as pre-authorization models in UCON, in which user-role assignments can be regarded as subject attributes, permission-role assignments can be regarded as object attributes, and the partial order relation between roles in role hierarchy is expressed by attribute predicates.

Example 12 Consider an RBAC1 model [Sandhu et al. 1996] where all roles R are in a partial order hierarchy with respect to domination relation $\geq$. A subject (a user in RBAC1) has an attribute $actRole$ with value a subset of R, the activated roles in a session. An object has an attribute $perRole$ with value a set of pairs $(role, r)$ where r is a right. A permission

(o, r) is assigned to a $role$ iff $(role, r) \in o.perRole$. The predicate $rpa_r(role, o)$ is true if there exists $role'$ such that $role \geq role'$ and $(role', r) \in o.perRole$.

The usage control policy for RBAC1 is expressed by:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (role \in s.actRole) \wedge rpa_r(role, o)\big)$

This is a basic $preA_0$ policy specifying that if $role$ is in the subject's $actRole$ attribute and $rpa_r(role, o)$ is true, then the subject can be granted access to the object with the right r.

RBAC with constraints can also be expressed with a UCON model. There are many types of constraints that can be defined in RBAC, such as mutually exclusive roles, cardinality, prerequisite roles, etc. [Sandhu et al. 1996]. With appropriate attributes defined for subjects and objects, we can specify RBAC models with constraints using UCON.

Example 13 Consider an RBAC2 model with an exclusive constraint, where $role_1$ can be activated by a user only if $role_3$ is not activated in the same session. Each object has the same attributes defined in the previous example. For each subject, besides the attribute $actRole$, the attribute $asgRole = \{role_1, role_2, \ldots, role_n\}$ denotes explicit user-role assignments. We can express this model in UCON $preA_1$ as follows:

(1) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (role_1 \in s.asgRole) \wedge (role_1 \notin s.actRole) \wedge (role_3 \notin s.actRole) \wedge rpa_r(role_1, o)\big) \wedge \blacklozenge preupdate(s.actRole)$
$preupdate(s.actRole) : s.actRole' = s.actRole \cup \{role_1\}$

This rule specifies that the permission (s, o, r) can be granted if $role_1$ is in the subject's $asgRole$ but not in $actRole$ (i.e., $role_1$ is assigned to s but not activated), $rpa_r(role_1, o)$ is true, and $role_3$ is not in the value of the attribute $actRole$ of the subject. The $permitaccess$ action implies a pre-update action of the subject's $actRole$ attribute by adding $role_1$ to it.

10.2 Chinese Wall Policy

The original Chinese Wall policy [Brewer and Nash 1988] prevents information flow between companies in conflict of interest. More generally, if a subject accesses an object in a conflict-of-interest set, then this subject cannot access any other object in this set in the future. We define an attribute to store the usage history of a subject: each time this subject generates an access request to an object, this attribute is checked and the authorization decision is determined by the history. In the meantime this attribute is updated to record this access information if the access request is approved. We show the policy with the following example.

Example 14 Consider a system with a set of conflict object classes $C = \{c_1, c_2, \ldots, c_n\}$. An object attribute $class$ indicates which class it belongs to. A subject attribute is defined as $ac = \{c_{s_1}, c_{s_2}, \ldots, c_{s_m}\}$, where $s_1, \ldots, s_m$ are integers from 1 to n, to record the classes that a subject has accessed. Another subject attribute is $ao = \{o_1, o_2, \ldots, o_k\}$, which stores the objects that the subject has accessed. If a subject has accessed an object, the Chinese Wall policy is:

(1) $permitaccess(s, o, read) \rightarrow \blacklozenge\big(tryaccess(s, o, read) \wedge (o \in s.ao)\big)$

For an access request for an object not in the subject's ao, the policy is:

(1) $permitaccess(s, o, read) \rightarrow \blacklozenge\big(tryaccess(s, o, read) \wedge (o \notin s.ao) \wedge (o.class \notin s.ac)\big) \wedge \blacklozenge preupdate(s.ac) \wedge \blacklozenge preupdate(s.ao)$

$preupdate(s.ac) : s.ac' = s.ac \cup \{o.class\}$
$preupdate(s.ao) : s.ao' = s.ao \cup \{o\}$

The first one is a $preA_0$ policy, which specifies that when a subject wants to access an object accessed before, the access request is approved and there is no update. The second one is a $preA_1$ policy because of the update of the subject's attributes. Specifically, if an object's conflict set is not in a subject's ac list, this subject can access this object, and both ac and ao must be updated before the access. Note that in this system there are two policies for the permission $(s, o, read)$. In a real access period, only one of them is satisfied, as we mentioned in Section 6.1.

10.3 Dynamic Separation of Duty

Dynamic separation of duty (DSoD) is a basic access control policy in many security systems. The concept of mutability for exclusiveness [Park et al. 2004] is presented to capture the attribute mutability property in DSoD. Specifically, an object attribute is defined to store the history of the subjects accessing this object. Here we present a simple example of object-based DSoD from [Simon and Zurko 1997].

Example 15 In a check issuing system, a check is prepared by a subject in the $clerk$ role and issued by a subject in the $supervisor$ role. A subject may have both a $clerk$ role and a $supervisor$ role at the same time, but a subject is not allowed to issue a check that is prepared by himself. For each object, the two attributes $preparer$ and $issuer$ store the subjects that prepare and issue this object, respectively. Initially the values of $preparer$ and $issuer$ are both $null$ (not available). Each subject has two attributes: sid (subject identity) and $role$. A predicate $\geq$ is defined to specify the dominance relation between two roles. The policies for $prepare$ and $issue$ are specified as follows, respectively.

(1) $permitaccess(s, o, prepare) \rightarrow \blacklozenge\big(tryaccess(s, o, prepare) \wedge (s.role \geq clerk) \wedge (o.preparer = null)\big) \wedge \blacklozenge preupdate(o.preparer)$
$preupdate(o.preparer) : o.preparer' = s.sid$

(2) $permitaccess(s, o, issue) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, issue) \wedge s.role \geq supervisor) \wedge (o.preparer \neq null) \wedge (o.issuer = null) \wedge (o.preparer \neq s.sid)\big) \wedge \blacklozenge preupdate(o.issuer)$
$preupdate(o.issuer) : o.issuer' = s.sid$

Both policies are $preA_1$ ones. The first one says that a subject with a role dominating $clerk$ can prepare a check, and this check's $preparer$ attribute is set to the subject's identity. The second one specifies that a subject with a role dominating $supervisor$ can issue a check only if this subject is not the one who prepares this check.

10.4 MAC Policy with High Watermark Property

In traditional MAC, a subject's clearance is assigned by a system administrator, and cannot be changed unless the administrator assigns a new label to it. This can be expressed with a UCON $preA_0$ model as shown in Section 6. With the high watermark property, the security clearance can be updated as a result of the user's access actions, and this update has to follow some predefined policies. We show this property in MAC as a $preA_1$ model.

Example 16 Suppose L is a lattice of security labels with relation $\geq$. A subject has two attributes, $clearance$ to represent the current label, and $maxClear$ to represent the

maximum clearance label. An object has one attribute, $classification$. All these attributes have as value domain the lattice L. The authorization policy for $read$ is:

(1) $permitaccess(s, o, read) \rightarrow \blacklozenge\big(tryaccess(s, o, read) \wedge (s.maxClear \geq o.classification)\big) \wedge \blacklozenge preupdate(s.clearance)$
$preupdate(s.clearance) : s.clearance' = LUB(s.clearance, o.classification)$

where LUB is a function that returns the least upper bound of two labels.

10.5 Hospital Information Systems

In this section we show some examples of hospital information systems that require not only authorizations, but also obligations and conditions.

Example 17 Suppose that a doctor (s) can perform (r) a particular operation (o) only if he has operated more than 3 times before[5]. This can be expressed as a $preA_1$ model. The total times of the operations that a doctor has performed is stored as the subject attribute exp. The policy is:

(1) $permitaccess(s, o, perform) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, perform) \wedge (s.role = doctor) \wedge (s.exp > 3)\big) \wedge \blacklozenge preupdate(s.exp)$
$preupdate(s.exp) : s.exp' = s.exp + 1$

Example 18 In this example, a doctor can perform an operation on a patient only if the patient agrees to it on a consent form. This agreement is an obligation to be completed before the operation, where the patient is the obligation subject, and the consent is the obligation object. This model can be expressed by a combination of $preA_0$ and $preB_0$. The policy is:

(1) $permitaccess(s, o, operate) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, operate) \wedge (s.role = doctor) \wedge \blacklozenge ob\_agree(o, consent)\big)$

The pre-decision components of this policy are a conjunction of an authorization predicate and an obligation, both of which must be satisfied before the access can start.

Example 19 In this example, a junior doctor can perform an operation only when there is a senior doctor monitoring the operation. An ongoing obligation $ob\_monitor(s_1, s_2)$ is defined where s_1 is a senior doctor and s_2 is a junior doctor. This model is a combination of $preA_0$ and onB_0.

(1) $permitaccess(s, o, operate) \rightarrow$
$\blacklozenge\big(tryaccess(s, o, operate) \wedge (s.role = junior\_doctor)\big)$

(2) $\square\big(\neg((s_1.role = senior\_doctor) \wedge ob\_monitor(s_1, s)) \wedge (state(s, o, operate) = accessing) \rightarrow revokeaccess(s, o, operate)\big)$

11. RELATED WORK

Bertino et al. [Bertino et al. 1994; Bertino et al. 1996; 1999] introduce a temporal authorization model for database management systems. In this model, a subject has permissions

[5]The examples in this section just show applications of our logical specification language, but do not provide a complete system specification. In this example, some other attribute predicates or conditions may enable a doctor to perform an operation at the beginning (when $exp \leq 3$), e.g., in the presence of senior doctors, which are not included here.

on an object during some time intervals, or a subject's permission is temporally dependent on an authorization rule. For example, a subject can access a file only for one week. Our authorization model is different: we consider the temporal characteristics in a single usage period, with mutable attributes of subject and object before, during and after an access, that is, the temporal properties are the result of the mutability of subject and object attributes, which change due to the side-effects of accesses and usages. In contrast, Bertino et al.'s model focuses on the validity of authorization policies with time period, and the temporal property of a policy is not related to an access action, but dependent on the system administration policies. Gal et al. [2000] propose a temporal data authorization model (TDAM) for access control to temporal data. This work is orthogonal to our approach, since we focus on the temporal authorization and usage process, while TDAM focuses on the temporal attributes of data. For formal specifications with temporal logic in security policies, Siewe et al. [2003] apply interval temporal logic to express and compose access control polices, and Hansen and Sharp [2003] introduce an approach for the analysis of security protocols using interval logic. The main difference in our approach is that we focus on the atomic actions and temporal properties during a single usage process, while their approaches focus on a higher level of system policies or security protocols.

Joshi et al. [2005] presented a generalized temporal RBAC model (GTRBAC) to specify temporal constraints in role activation, user-role assignment, and role-permission assignment. For example, a user can only activate a role for a particular duration. The concept of temporal constraint is different from the mutability of UCON since it does not have update actions. The dependency constraint in GTRBAC [Joshi et al. 2003] is similar to the concept of obligation in UCON, but the dependency is more like the implication relation between events in GTRBAC, i.e., if an event happens, it triggers another event; while in UCON, obligations are explicit required actions to permit an access.

Bettini et al. [2002a; 2002b] present concepts of provisions and obligation in policy management: provisions are conditions or actions performed by a subject before the authorization decision, while obligations are conditions or actions performed after an access. In our model, we distinguish between conditions and obligations. All the actions that a subject has to perform before usage are regarded as obligations, while for future actions, we consider them as the obligations for future usage requests or long-term obligations. Chomicki and Lobo [2001] investigate the conflicts and constraints of historical actions in policies. In their paper, actions are application activities, and constraints are expressed with linear-time temporal connectors. In our paper we define obligations as actions required by an access, and represent the logic approach with TLA.

12. CONCLUSIONS AND FUTURE WORK

We have developed a formal specification of UCON with temporal logic of actions. A logic model is given by a set of sequences of system states specified by a set of subjects and their attributes, a set of objects and their attributes, and the system attributes. The authorization predicates are built from subject and object attributes. Actions are the state transitions of the system, including usage control actions to update attributes and accessing status of a usage process, and obligation actions that have to be satisfied before or during an access. Conditions are predicates on system attributes. Temporal formulae represent usage control policies and are built from authorization predicates, actions, and system predicates. We prove that a fixed set of scheme rules can be used in UCON specifications with soundness

and completeness properties. The powerful specification capability and flexibility of the extended TLA strengthens UCON with precise modeling and specification.

This work opens several directions for further investigation. First of all, we can develop administrative models for UCON, including attributes management, administrative policies, etc. UCON is attribute-based, and this requires synchronized attribute acquisition and management. As mentioned in this paper, post-obligations and post-conditions are in the scope of the administrative model. If a subject does not satisfy an obligation after an access, a security administrator needs to take compensatory actions according to the administrator policies.

As a comprehensive access control model with new properties, UCON has shown strong expressivity and flexibility to specify modern access control systems. In general, the expressive power and the decidability of safety are two conflicting properties of an access control model. We are investigating the safety problem, which is a fundamental problem in access control. In UCON, the safety problem consists of deciding whether a subject can obtain a particular permission on an object, given a set of attributes and initial values, as well as updates of these attributes by performing some accesses. Restricted UCON models with reasonable expressive power and decidable safety are under investigation.

As mentioned in Section 1, concurrency is a unique feature in UCON which has been seldom investigated in access control models. In an open system, an update action of an attribute will result in a change in the authorization decision in another access happening concurrently. The properties of access control models in such open and concurrent environments need to be investigated.

REFERENCES

BELL, D. E. AND LAPADULA, L. J. 1975. Secure computer systems: Mathematical foundations and model. *Mitre Corp. Report No.M74-244, Bedford, Mass.*.

BERTINO, E., BETTINI, C., FERRARI, E., AND SAMARATI, P. 1996. A temporal access control mechanism for database systems. *IEEE Transactions on Knowledge and Data Engineering 8*, 1 (Feb.).

BERTINO, E., BETTINI, C., FERRARI, E., AND SAMARATI, P. 1999. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transaction on Database Systems 23*, 3 (Sept.).

BERTINO, E., BETTINI, C., AND SAMARATI, P. 1994. A temporal authorization model. In *Proceedings of ACM Conference on Computer and Communication Security*. ACM.

BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. 2001. A logical framework for reasoning about access control models. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*. ACM.

BETTINI, C., JAJODIA, S., WANG, X. S., AND WIJESEKERA, D. 2002a. Obligation monitoring in policy management. In *Proceedings of the 3rd Internationl Workshop on Policies for Distributed Systems and Networks*.

BETTINI, C., JAJODIA, S., WANG, X. S., AND WIJESEKERA, D. 2002b. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th VLDB Conference*.

BREWER, D. AND NASH, M. 1988. The chinese wall security policy. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*.

CHOMICKI, J. AND LOBO, J. 2001. Monitors for history-based policies. In *Proceedings of the 2nd Internationl Workshop on Policies for Distributed Systems and Networks*.

DAMIANOU, N., DULAY, N., LUPU, E., , AND SLOMAN, M. 2001. The ponder policy specification language. In *Proceedings of the Workshop on Policies for Distributed System s and Networks*.

DENNING, D. E. 1976. A lattice model of secure information flow. *Communications of the ACM 19*, 5 (May).

FERRAIOLO, D. F., SANDHU, R., GAVRILA, S., KUHN, D. R., AND CHANDRAMOULI, R. 2001. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security 4*, 3.

GAL, A. AND ATLURI, V. 2000. An authorization model for temporal data. In *Proceedings of the ACM Conference on Computer and Communication Security*.

HANSEN, M. AND SHARP, R. 2003. Using interval logics for temporal analysis of security protocols. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*.

HARRISON, M. H., RUZZO, W. L., AND ULLMAN, J. D. 1976. Protection in operating systems. *Communication of ACM 19*, 8.

JAJODIA, S., SAMARATI, P., , AND SUBRAHMANIAN, V. S. 1997. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*.

JAJODIA, S., SAMARATI, P., SAPINO, M. L., AND SUBRAHMANIAN, V. S. 2001. Flexible support for multiple access control policies. *ACM Transactions on Database Systems 26*, 2 (June).

JOSHI, J., BERTINO, E., LATIF, U., AND GHAFOOR, A. 2005. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering 17*, 1.

JOSHI, J., BERTINO, E., SHAFIQ, B., AND GHAFOOR, A. 2003. Constraints: Dependencies and separation of duty constraints in gtrbac. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May).

MANNA, Z. AND PNUELI, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer-Verlag.

PARK, J. AND SANDHU, R. 2004. The $UCON_{ABC}$ usage control model. *ACM Transactions on Information and Systems Security 7*, 1 (Feb.).

PARK, J., ZHANG, X., AND SANDHU, R. 2004. Arrtibute mutability in usage control. In *Proceedings of the Proceedings of 18th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*.

SANDHU, R. 1992. The typed access matrix model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*.

SANDHU, R. 1993. Lattice-based access control models. *IEEE Computer 26*, 11 (Nov.).

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role based access control models. *IEEE Computer 29*, 2.

SANDHU, R. AND PARK, J. 2003. Usage control: A vision for next generation access control. In *Proceedings of the Second International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security*.

SIEWE, F., CAU, A., AND ZEDAN, H. 2003. Compositional framework for access control policies enforcement. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*.

SIMON, R. T. AND ZURKO, M. E. 1997. Separation of duty in role-based environments. In *IEEE Computer Security Foundations Workshop*.