Logical Model and Specification of Usage Control

XINWEN ZHANG
FRANCESCO PARISI-PRESICCE
JAEHONG PARK
RAVI SANDHU
George Mason 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 first-order logic specification of UCON with an extension of Lamport's temporal logic of actions (TLA). The building blocks of this model include: (1) a sequence of states expressed by the attributes of the subjects, the objects, and the system, (2) authorization predicates 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 on system attributes. Usage control policies are defined as a set of temporal logic formulas that are satisfied as system state changes. We show the flexibility and expressive capability of this logic model by specifying the core models of UCON and some applications. We also show how model checking based on computational tree logic can verify security policies in a UCON system specified in this manner.

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, logic specification

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] primarily consider static authorization decisions based on subjects' permissions on target objects. Policy-based authorization management systems have been proposed [Bertino et al. 2001; Damianou et al. 2001; Jajodia et al. 2001; Jajodia et al. 1997], in which a centralized reference monitor (or distributed reference monitor with centralized

This research was partially supported by the National Science Foundation grant CNS-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: 4400 University Drive, George Mason University, MSN 4A4, Fairfax, VA 22030.
Email: {xzhang6, fparisip, jpark2, sandhu}@gmu.edu
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2000 ACM 1094-9224/2000/1100-0111 $5.00

administration) checks a subject's permission when access is requested, and the request is granted according to the security policies at the time of the access request. Once a subject is granted a permission, the object can be accessed repeatedly.

The development in information technology, especially in electronic commerce applications, requires additional features for access control. In recent information systems, the usage of a digital object can not only be one instant 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.

Recently proposed usage control (UCON) is a new access control model that extends traditional access control models in multiple aspects [Park and Sandhu 2004]. 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 using attributes of the subject, object, and right. Obligations are actions that are required to be performed before or during the access process. Conditions are environment restrictions that are required to be valid before or during access. An extreme example of UCON is the traditional access control models, in which the authorization decision is typically 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 mechanism of existing access control models and policies, as well as the access control in digital rights management (DRM), trust management, and other modern information systems.

The distinguishing properties of UCON beyond traditional access control models are continuity of access decision and mutability of subject attributes and object attributes. In UCON, authorization decisions are not only checked and made before the 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 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 authorization, obligations, and conditions. The main contribution of this paper with respect to previous papers is that we formalize UCON model with first-order 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 designer and administrator. With a conceptual and informal model, the capability to define policy is limited. Also, a logical specification provides precise meaning of new features of UCON, such as mutability of attributes and continuity of usage control decisions. Further, the verification of security properties in a specific UCON system needs a logical formalization of the model. Finally, to analyze 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.

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. The mutability property of UCON is illustrated in Section 4. Section 5 introduces TLA briefly. Section 6 presents the details of our logic model. Section 7 presents the specification of the core UCON authorization models with our logic model. Section 8 and Section 9 introduce the logical specification of obligation core models and condition core models, respectively. Section 10 illustrates the flexibility and expressive power of our logical model. Section 11 discusses the security verification with model checking mechanisms in TLA specified UCON models. Some related work in access control with temporal aspects is reviewed in Section 12. Finally, we summarize this paper and present some ongoing and future work in Section 13.

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 user with the longest accumulating usage time is revoked.

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

(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, the set of accessing subjects 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 ongoing 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 total accessing number is updated by subtracting one, and the subject is removed from the accessing list. This is called a post-update.
(b) Objects have the same attributes as in (a). Each subject has two attributes: the status of the subject with a value of $busy$ or $idle$, and 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 objects have the same attributes 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 of 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. There are many other examples to motivate UCON model that cannot be expressed by traditional access control models. We explore some of these later in this paper as we describe our logical approach. An access decision is no longer a single function of (subject, object, right), but may depend on attributes of the entities involved in the access, and may change the attributes of these entities. On the other side, an access is not a simple action, but consists of a serious of actions and events not only from a subject, but also from the system.

3. USAGE CONTROL

In this section we briefly review the general ideas of UCON and the core authorization models. The details of these models can be found in [Sandhu and Park 2003; Park and Sandhu 2004].

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

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 rule permits or denies access of a subject to an object with a specific right based on subject and object attributes. Obligations are activities that have to be performed by subjects before or during an access. Conditions are system environment restrictions, not related to subject or object attributes.

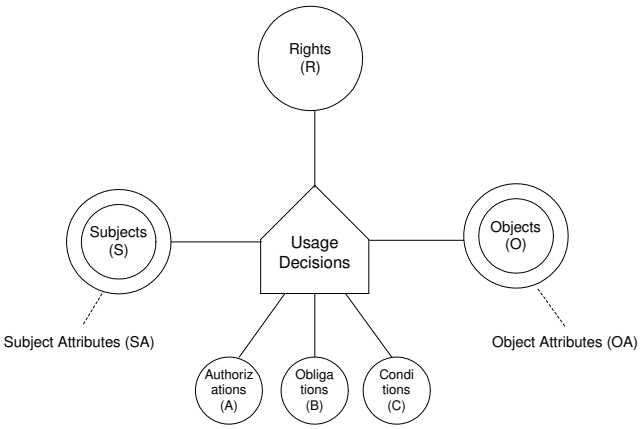

Fig. 1. Usage control model

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 a control policy may be enforced not only before an access, but 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 are checked and enforced in the first two phases, named pre-decisions and ongoing-decisions respectively. In the after-usage phase, we don't enforce any policy since there is no access control after a subject finishes a usage on an object[3].

Mutability means that subject or object attribute may be updated to a new value as a result of accessing. 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 subject or object attributes may result in a system action to permit or revoke an access. An update can affect not only the concurrent usage, but also other usages related to the same subject or object. An update on the current usage may generate

[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.

[3]There can be obligations and conditions (post-obligation and post-conditions) defined in this phase. UCON is a session-based access control model, since it controls the current access request and ongoing access. The obligations and conditions after an access are regarded as long-term obligations and conditions, which are not included in the core UCON, 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.

cascading updates, while an update on other usages can act as external events that would cause a change of the concurrent usage, such as revocation.

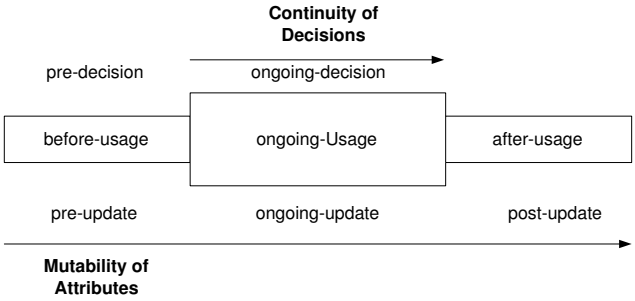

Fig. 2. Continuity and mutability properties of UCON

4. ATTRIBUTE MANAGEMENT AND MUTABILITY

Usage control model includes several underlying assumptions. In UCON, usage decision is request-based, i.e., rights are not pre-assigned to subjects and permissions are granted at the time of usage requests. Authorization decisions are based on subject attributes and object attributes according to usage control policies. Depending on 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 *"admin-controlled"* or *"system-controlled"*. This section discusses these two categories. Figure 3 shows a taxonomy of attribute management.

4.1 Admin-controlled Attribute Management (Immutable)

Administrator-controlled attributes can be modified only by explicit administrative actions. These attributes are modified at the administrator 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

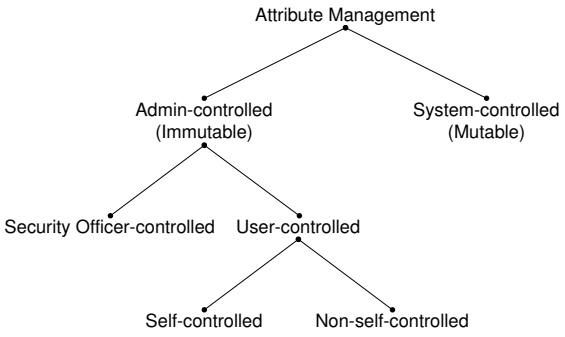

Fig. 3. Attribute management taxonomy

are made by security officers. If a subject is assigned to a new security label or to a new membership group because of a management decision, updates on attributes are made by administrative actions. This is a typical approach in traditional access control policies such as MAC and RBAC. Static separation of duty and user-role assignment in RBAC belongs in this category. However, there are other cases where subject attributes are controlled by a user. This *user-controlled* attribute management can be further classified into *self-controlled* and *non-self-controlled*. An example of self-controlled attribute management is role activation in RBAC, where a user can activate or deactivate his or her roles in a session. Controlling a users' ability to update attributes (e.g., activated roles) is also considered as an administrative issue. In non-self controlled cases, attributes are controlled by a user other than the user of subjects or sessions. For example, in an online music store, the parents of a child may preset the child's maximum purchase limits to 20 dollars a month by controlling the attributes of the child. In UCON, all these cases are considered as part of the administrative model and are not included in this paper.

4.2 System-controlled Attribute Management (Mutable)

Unlike admin-controlled, in system-controlled attribute management, updates are made as side effects or results of user'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 admin-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. Therefore attribute mutability is considered as part of UCON core models. In both admin-controlled and system-controlled management, it is the security officer who manages the ability of user updates and system updates. In this paper our concern lies in the system-controlled mutability issue where updates are made as side effects of users' 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.

5. 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.

5.1 Building Blocks

Variables, values, and states are basic concepts in TLA. Values areelements 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 a fixed value. A state is characterized by assignment of a value $s[\![x]\!]$ to each variable x.

A function is a nonboolean 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. Generally:

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

where $f(\forall \text{`}v\text{'}: s[\![v]\!]/v)$ is the value obtained by substituting $s[\![v]\!]$ for v. Semantically, 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 booleans. 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 $x' - 1 \notin y'$. Semantically, 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:

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

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 $true$.

5.2 Temporal Formulas 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:

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

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:

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

5.3 Extension of TLA

Other future operators, such as "eventually" ($\Diamond$) and "infinitely often" ($\Box\Diamond$) 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 formulas, we can define other temporal operators and semantics similarly.

5.3.1 *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. Semantically:

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

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:

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

5.3.2 *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 past time compared to 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 series. We use the state sequence $..., s_{-2}, s_{-1}$ for states during the past time along this time series. Therefore, a behavior is a state sequence:

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

Based on this, we can define past temporal operators similar to the future ones: $\blacksquare$ (*Has-always-been*), $\blacklozenge$ (*Once*), $\ominus$ (*Previous*), $\mathcal{S}$ (*Since*). Semantically:

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

Similar to the future operators, there are some equivalences among these past operators. For example,

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

6. 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.

6.1 Attributes and States

In TLA, a 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 the state of each subject or object is specified by a finite set of attributes. We require that each entity (subject or object) 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. Generally, $ent.a \in ATT(ent)$, where $ATT(ent)$ is a finite set of attributes for each entity ent.

A subject or object attribute is a variable of a specific datatype, which includes a set of possible values and operators to manipulate them. A state of a subject or an object is an assignment of values to attribute(s). 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 $ent.att = value$. Sometimes we just use $ent.att$ to denote an attribute value when the context is clear.

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 state 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 decision; $denied$ means that the system has denied the access request according to the authorization 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 will go to $revoked$ state when its ongoing-access is revoked by the system, or it will go 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. Semantically, 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 subject attributes, object attributes, and system attributes.

6.2 Predicates

A predicate is a boolean expression built from variables and constants, where variables includes subject attributes, object attributes, and system attributes. The semantics of a predicate is a mapping from states to boolean values. A state satisfies a predicate if the attribute values assigned in this state satisfy the predicate. For example, the predicate $Alice.credit > \$100.0$ is $true$ if Alice's $credit$ attribute value in the current state of the system is larger than \$100.0. Since a system may have very different predicates from another system, the set of predicates for a general UCON$_A$ model is not fixed. A unary predicate is built from one attribute variable and constants, e.g., $Alice.credit \geq \$100.00$, $file1.classification = `supersecure'$. A binary predicate is built from two different attribute variables, e.g., $dominate(Alice.cleareance, file1.classification)$, $Alice.credit \geq$

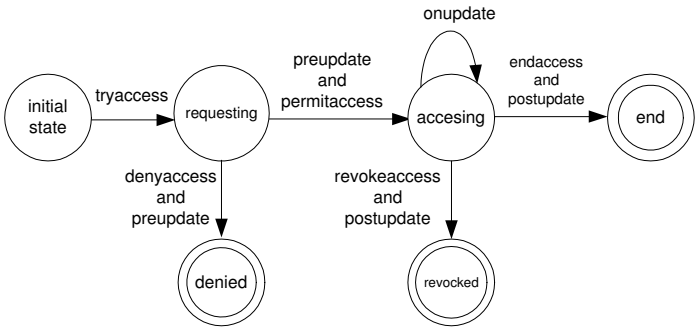

Fig. 4. State transition of a single access with usage control actions

$ebook.value$, $(Alice, r) \in file1.acl$, where $file1.acl$ is object $file1$'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. There is a special predicate $permit(s, o, r)$ in a UCON system, which is $true$ if a subject s can access an object o with right r. This predicate is the result of a usage control decision evaluated by the system.

6.3 Actions

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

6.3.1 *Usage Control Actions.* Usage control actions include actions to update attribute values, and actions to change the state of a single access process. 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, as the changes of system attributes are not captured in the core models.

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

If the system performs the action successfully, the attribute value is changed to a new value, and the action is $true$, otherwise, it is $false$. Note that in our specification we do not consider the time delay of an action, and we assume that an action is always performed instantly causing the transition to the next state. In a real implementation, there is a mechanism to monitor the process and audit the update, so that the system can recover in the event of a failure.

Other usage control actions are performed by a subject or the system and can 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 4. Note that Figure 4 only shows the state transitions of the system attribute $state(s, o, r)$ in one usage session. It does not show subject/object attributes or other system attributes, usually included in the state.

We can categorize all the usage control actions into two classes: actions performed by a subject and actions performed by the system. Figure 5 shows these actions during the life cycle of a usage, which are briefly explained below.

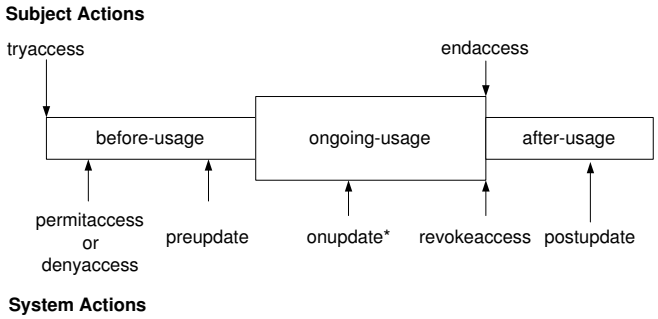

Fig. 5. Usage control actions

(1) $tryaccess(s, o, r)$: generates a new access request (s, o, r), performed by a subject.
(2) $permitaccess(s, o, r)$: grants an access request of (s, o, r), performed by the system.
(3) $denyaccess(s, o, r)$: rejects an 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 a subject.
(6) $preupdate(attribute)$: updates a subject or object attribute before access, performed by the system.
(7) $onupdate(attribute)$: updates a subject or object attribute during the usage phase, performed by the system. The star symbol in Figure 5 indicates that this action may be performed repeatedly by the system to continuously update an attribute.
(8) $postupdate(attribute)$: updates a subject or object attribute after access, performed by the system.

6.3.2 *Obligation Actions.* In UCON an obligation is an action that must be performed by a subject before or during an access. Formally, an obligation is a statement with variables and attributes between two system states. In this paper we do not explicitly include any variables or attributes from which obligations are built, since the obligation requirements of an access request heavily depend on specific applications. But implicitly, each obligation involves one or more attributes or variables between two system states.

Definition 6.1. An obligation is an action described by $ob(s, o, r, s_b, o_b)$ where ob is the action name, (s, o, r) is a particular usage process requiring the obligation, and s_b, 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 the particular application. For example, the downloading of a music file may need the action to click the privacy button by the same subject. The obligation is defined as

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

where the obligation subject is the same as the accessing subject, and $privacy\_statement$ is the obligation object. For another example, a child's watching an online movie may need the parents' agreement in advance. In this paper we assume that an obligation action is always doable whenever required, so that an obligation is not dependent on other permissions or attribute predicates.

6.4 Model and Satisfactions of Formulas

Definition 6.2. 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 sequence of states of the system,

—$\mathcal{P}_\mathcal{A}$ 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 logic formula is built from predicates and actions with logic connectors and temporal operators.

Definition 6.3. A logical formula in UCON is defined by the following grammar in BNF:

$$\phi ::= a | p(t_1, ..., t_n) | (\neg\phi) | (\phi \wedge \phi) | (\phi \rightarrow \phi) | \Box\phi | \Diamond\phi | \bigcirc\phi | \phi\mathcal{U}\phi | \blacksquare\phi | \blacklozenge\phi | \ominus\phi | \phi\mathcal{S}\phi |$$

where a is an action, p is a predicate of arity n, and t_1, ..., t_n are terms.

If a model $\mathcal{M}$ with a state s satisfies a formula ϕ, we write $\mathcal{M}, s \vDash \phi$. Formally,

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

7. SPECIFICATION OF UCON 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 TLA-based 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. Park and Sandhu [Sandhu and Park 2003; Park and Sandhu 2004] defined seven core authorization models summarized below.

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

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

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

Note that in the case of authorization before access and update during usage, since the update of attributes does not trigger any authorization check during usage, it has the same effect as update after usage ($preA_3$)[4]. So this case is not included in UCON. For models which enforce authorizations during a usage, ongoing-checking captures not only the attribute changes from this local usage process, but also other related usage processes. For example, a subject's attribute change due to the system administrator's action may revoke his ongoing access to an object if the authorizations of this access is no longer valid.

7.1 The Model $preA_0$

As presented in [Sandhu and Park 2003; Park and Sandhu 2004], most traditional access control models can be expressed in $preA_0$ 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 policies are:

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

where p_1, ..., p_i are state predicates built from subject and/or object attributes. The $permit$ predicate states that s can access o with r. The $permitaccess$ action grants the permission to s and starts the access. Since there is no update between the access request and the grant action, $permitaccess$ is true in the "next" state of $tryaccess$.

[4]This assumes no "interference" with other ongoing accesses.

Example 1 In mandatory access control (MAC), each subject is assigned to a security clearance, and each object is assigned to 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 some 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 can be expressed in UCON as a $preA_0$ model as shown below.

$$dominate(s.clearance, o.classification) \rightarrow permit(s, o, read)$$
$$tryaccess(s, o, read) \wedge permit(s, o, read) \rightarrow \bigcirc\big(permitaccess(s, o, read)\big)$$
$$dominate(o.classification, s.clearance) \rightarrow permit(s, o, write)$$
$$tryaccess(s, o, write) \wedge permit(s, o, write) \rightarrow \bigcirc\big(permitaccess(s, o, write)\big)$$

$dominate$ is a binary predicate on subject attributes and object attributes, 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 $preA_0$ model. A subject attribute is its identity, and an object attribute is an access control list acl of a set of pairs of (id, r), where id is a subject's identity, and r is a right with which this subject can access this object.

$$(s.id, r) \in o.acl \rightarrow permit(s, o, r)$$
$$tryaccess(s, o, r) \wedge permit(s, o, r) \rightarrow \bigcirc\big(permitaccess(s, o, r)\big)$$

7.2 The Model $preA_1$

In $preA_1$, the authorization rules are checked before the access, and there are one or more update actions before the system grants the permission to a subject. The usage control policies are:

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

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, an access request must have occurred before, the $permit$ predicate must have been $true$, and there was a $preupdate$ action that occurred after that. In this formula, the $permit$ predicate is only required to be $true$ before the action $preupdate$. Note that only one update action is specified here. This assumption is applied for other core models in this paper. We will explicitly mention when multiple updates are needed.

We assume that the time line is bounded during the life time of an access period. The "Once" operator does not refer to any past action before $tryaccess$. The same assumption is also made for future temporal operators.

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 policies are:

$$(Alice.credit \geq ebook1.value) \rightarrow permit(Alice, ebook1, read)$$
$$permitaccess(Alice, ebook1, read) \rightarrow \blacklozenge\big(tryaccess(Alice, ebook1, read) \wedge$$

$$permit(Alice, ebook1, read) \wedge \Diamond(preupdate(Alice.credit))\big)$$
$$preupdate(Alice.credit) : Alice.credit' = Alice.credit - ebook1.value$$

The first rule specifies that whenever Alice's credit is more than the value of ebook1, she can read it. The second rules says that 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.

7.3 The Model $preA_3$

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

$$p_1 \wedge ... \wedge p_i \rightarrow permit(s, o, r)$$
$$permitaccess(s, o, r) \rightarrow \blacklozenge(tryaccess(s, o, r)) \wedge permit(s, o, r)$$
$$endaccess(s, o, r) \rightarrow \Diamond(postupdate(attribute))$$

The first two rules are the same as before, except that the update action does not appear in the second rule. Note that the second rule is the same as

$$permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge permit(s, o, r)\big)$$

since there is no update in the before-usage phase. Actually, since $permit(s, o, r)$ is only dependent on the attributes of s and o, and there is no update before the $endaccess$ action, once the access is approved, $permit(s, o, r)$ must be true from the $tryaccess$ action to the $endaccess$ action. The third rule says that a $postupdate$ action must be performed by the system after an access is ended by a subject. Since the authorization rules are not enforced after granting the access, there is no access revocation in this model.

Example 4 In a DRM membership-based application, a reader s has attributes id and total $expense$, and a book o has attributes $title$ and $readingCost$. A reading group is a subject with attributes $readerList = \{id1, id2, ...\}$ and $bookList = \{book1.title, book2.title, ...\}$. The policies are:

$$(s.id \in readingGroup.readerList) \wedge (o.title \in readingGroup.bookList)$$
$$\rightarrow permit(s, o, read)$$
$$permitaccess(s, o, read) \rightarrow$$
$$\blacklozenge(tryaccess(s, o, read)) \wedge permit(s, o, read)$$
$$endaccess(s, o, read) \rightarrow \Diamond(postupdate(s.expense))$$
$$postupdate(s.expense) : s.expense' = s.expense + o.readingCost$$

In this example, the authorization policy states that if both s and o belong to $readingGroup$, then s can read the book and his expense is updated by adding the cost of this book after the access.

7.4 The Model onA_0

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

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

In the onA_0 model, the authorization predicates have to be satisfied in any state during the access period, otherwise the access is revoked by the system immediately. The policy says that, after the action $permitaccess$, the formula

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

must be always true, so that either the authorization predicates $p_1, ..., p_i$ are true when the subject is $accessing$ the object in a state, or there is an $revokeaccess$ action from the system in that state.

The policy in onA_0 model can also be specified as the following formula with *"Until"* operator:

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

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, then 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, we do not include the pre-authorization rules in ongoing-authorization models. In practice, real applications may require a combination of several core models.

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 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 control rule for the access is:

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

7.5 The Model onA_1

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

$$permitaccess(s,o,r) \rightarrow \blacklozenge\big(tryaccess(s,o,r) \wedge \Diamond(preupdate(attribute))\big)$$
$$permitaccess(s,o,r) \rightarrow \Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

Since there is no authorization check before a subject starts to access an object, in the first rule, the $permitaccess$ action implies only an update action before it.

7.6 The Model onA_2

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

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

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

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

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

7.7 The Model onA_3

In onA_3, there must be update action(s) after a usage process. The control policies are:

$permitaccess(s,o,r) \rightarrow \Box\big(\neg(p_1 \wedge ... \wedge p_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$
If an access is ended by the subject:
$endaccess(s,o,r) \rightarrow \Diamond(postupdate(attribute))$
If an access is revoked by the system:
$revokeaccess(s,o,r) \rightarrow \Diamond(postupdate(attribute))$

In many applications, the update after an access is ended by a subject, is different from the one after an access is revoked by the system. 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.

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 a 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 authorization rules are:

(1) **true** $\rightarrow permit(s,o,r)$
(2) $permitaccess(s,o,r) \rightarrow \blacklozenge\big(tryaccess(s,o,r) \wedge permit(s,o,r) \wedge \Diamond(preupdate(o.accessingS)) \wedge \Diamond(preupdate(s.startTime))\big)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$
$preupdate(s.startTime) : s.startTime' = sys.clock$
(3) $permitaccess(s,o,r) \rightarrow \Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s,o,r) = accessing) \wedge (s.startTime = Min_{startTime}(o.accessingS)) \rightarrow revokeaccess(s,o,r)\big)$
(4) $endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow \Diamond(postUpdate(o.accessingS) \wedge \Diamond(postUpdate(s.startTime)$
$postUpdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$
$postUpdate(s.startTime) : s.startTime' = null$

where $|o.accessingS|$ is the number of accessing subjects, and $Min_{startTime}(o.accessingS)$ is the earliest start time from $accessingS$. The first rule says that a new access request

is always permitted. The second rule is a $preA_1$ 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 local system's clock. The third rule says that when the total number of accessing user is larger than 10 (a new request is generated and approved), the subject with earliest start time is revoked. The fourth 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.

(b) Revocation by the longest idle time:
We define two subject attributes: the status of the usage ($status$ with value $busy$ or $idle$) and continuous idle time in a single usage period ($idleTime$). The control rules are:

(1) **true** $\rightarrow permit(s, o, r)$
(2) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge permit(s, o, r) \wedge \Diamond(preupdate(o.accessingS)) \wedge \Diamond(preupdate(s.idleTime))\big)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$
$preupdate(s.idleTime) : s.idleTime' = 0$
(3) $permitaccess(s, o, r) \rightarrow \Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s, o, r) = accessing) \wedge (s.idleTime = Max_{idleTime}(o.accessingS)) \rightarrow revokeaccess(s, o, r)\big)$
(4) $\Box\big((state(s, o, r) = accessing) \wedge (s.status = idle) \rightarrow onupdate(s.idleTime)\big)$
$onupdate(s.idleTime) : s.idleTime = s.idleTime + 1$
(5) $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow \Diamond(postupdate(o.accessingS))$
$postupdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$

where $Max_{idleTime}(o.accessingS)$ is the largest $idleTime$ in the object's $accessingS$ attribute. Rules (1), (2), and (5) are the same as before, except that in rule (2), one pre-update action is to initialize the subject's $idleTime$. In rule (3), the revocation is determined by the $s.idleTime$. Rule (4) 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 state of a subject is $idle$.

(c) Revocation by the longest total usage time:
We define the accumulating usage time $usageTime$ as a subject attribute. The control rules are:

(1) **true** $\rightarrow permit(s, o, r)$
(2) $permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge permit(s, o, r) \wedge \Diamond(preupdate(o.accessingS))\big)$
$preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$
(3) $permitaccess(s, o, r) \rightarrow \Box\big(\neg(|o.accessingS| \leq 10) \wedge (state(s, o, r) = accessing) \wedge (s.usageTime = Max_{usageTime}(o.accessingS)) \rightarrow revokeaccess(s, o, r)\big)$
(4) $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow \Diamond(postupdate(s.usageTime) \wedge \Diamond(postupdate(o.accesingS))$
$postupdate(o.accesingS) : o.accessingS' = o.accessingS - \{s\}$
$postupdate(s.usageTime) : s.usageTimes' = s.usageTime + sys.periodT$

where $Max_{usageTime}(o.accessingS)$ is the largest $usageTime$ in $accessingS$. Rules (1), (2) are the same as those in the previous case except that there is only one pre-update

action; rule (3) specifies that the revocation is determined by the total usage time of the subject. Rule (4) says that after each usage, there must be an update on $usageTime$ by adding this usage time to the old value. The $sys.periodT$ is a system attribute to record this accessing's period [5]. 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.

8. SPECIFICATION OF UCON 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 of 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 we need an administration model to capture the 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 future work on administrative models.

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

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

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

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

[5] 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 here.

8.1 The model $preB_0$

Similar to the model $preA_0$, the policies of $preB_0$ are:

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

The $ob_1, \ldots, ob_i$ are actions of obligations for access (s,o,r). The first rule requires that an access is enabled after all the obligations are satisfied. The difference between $preB_0$ and $preA_0$ is that in the former an obligation can be satisfied any time before an access is enabled, so that the "once" operator is applied in the first rule; while in the latter the authorization predicates must be satisfied at the moment of the $tryaccess$ action. Note that here we just ignore the authorization factors (attribute predicates), since we are discussing the 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: the obligation subject is the same as the ordering subject, while $agree\_statement$ is the obligation object. The usage control policies for this application are:

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

8.2 The Model $preB_1$

The policies for $preB_1$ are:

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

The first rule is the same as in $preB_0$. In the second rule, an update action must be performed after $tryaccess$ and before $premitaccess$. Note that this rule is different from that in $preA_1$, since here the $permit$ predicate is determined by obligations, which can be satisfied at any time before the $permitaccess$ action, while in $preA_1$, the $permit$ predicate depends on attribute predicates evaluated before the update action.

Example 8 In an online electronic marketing system, an anonymous user needs to register first in order to browse the products. The registration action $regist$ is a pre-obligation action, after which the user's role is changed from $anonymous$ to $registered$. This can be expressed in a $preB_1$ model as follows:

$$\blacklozenge regist(s,o,browse,s,system) \rightarrow permit(s,o,browse)$$
$$permitaccess(s,o,browse) \rightarrow permit(s,o,browse) \wedge$$
$$\blacklozenge\big(tryaccess(s,o,browse) \wedge \Diamond(preupdate(s.role))\big)$$
$$preupdate(s.role) : s.role' = registered$$

In this policy, $regist(s,o,browse,s,system)$ is an obligation action for access of a subject's browsing an object. The subject in the obligation is the same subject who generates the access request, while the object is a system object sys, on which a subject can take the reg right to be registered. The first rule specifies that once the registration obligation has been completed, the $permit$ predicate becomes true in the next state. Note that since there is only one obligation for this access, the first rule can be replaced by the following one:

$$\ominus\big(regist(s,o,browse,s,system,reg)\big) \rightarrow permit(s,o,browse)$$

This rules states that an access is enabled[6] once after the subject finishes his registration. The second formula specifies that before the system performs the $permitaccess$ action, there must be a $typeaccess$ action performed by the subject, the $permit$ predicate must be true, and $preupdate$ action has been completed. The $preupdate$ action updates the subject's role to a new value.

8.3 The Model $preB_3$

The policies for $preB_3$ are:

$$\blacklozenge ob_1 \wedge \blacklozenge ob_2 \wedge \ldots \wedge \blacklozenge ob_i \rightarrow permit(s,o,r)$$
$$permitaccess(s,o,r) \rightarrow permit(s,o,r) \wedge \blacklozenge\big(tryaccess(s,o,r)\big)$$
$$endaccess(s,o,r) \rightarrow \Diamond\big(postupdate(attribute)\big)$$

The rules in this policy are similar to those in $preA_3$, except that the $permit$ predicate is determined by a set of obligations and their satisfactions are evaluated when the $permitaccess$ action is performed.

8.4 The Model onB_0

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

$$permitaccess(s,o,r) \rightarrow \Box\big(\neg(ob_1 \wedge ... \wedge ob_i) \wedge$$
$$(state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

Similar to onA_0, the policy specifies that after the action $permitaccess$, the formula

$$\neg(ob_1 \wedge ... \wedge ob_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

must be always true during an accessing process, so that either the obligations $ob_1, ..., ob_i$ are true when the subject is $accessing$ the object, or the access is revoked immediately.

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.

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

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

8.5 The Model onB_1

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

$$permitaccess(s,o,r) \rightarrow \blacklozenge\big(tryaccess(s,o,r) \wedge \Diamond(preupdate(attribute))\big)$$
$$permitaccess(s,o,r) \rightarrow \Box\big(\neg(ob_1 \wedge ... \wedge ob_i) \wedge$$
$$(state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

[6]Here "enabled" means that the $permit$ predicate is true. The real permission is not granted until a request is generated and the corresponding update(s) has been performed based on the second rule.

The first 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, in the first rule, the $permitaccess$ action implies only a $tryaccess$ and a $preupdate$ action before it. The second rule is the same as that in onB_0.

8.6 The Model onB_2

In onB_2, there are one or more update actions during an accessing process. The policies are:

$$permitaccess(s,o,r) \rightarrow \Box\big(\neg(ob_1 \wedge ... \wedge ob_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)\ endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow \blacklozenge\big(permitaccess(s,o,r) \wedge \Diamond(onupdate(attribute))\big)$$

In the second rule, we only specify that there is an update action. For the cases where an update is required in every state change during the ongoing access, the second rule becomes:

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

8.7 The Model onB_3

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

$$permitaccess(s,o,r) \rightarrow \Box\big(\neg(ob_1 \wedge ... \wedge ob_i) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$$

If the access is ended by subject:

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

If the access is revoked by system:

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

Note that the update after an access is ended by a subject is usually different from the one after an access is revoked by the system. Here, we simply use the same action name of $postupdate$, but they may change the attribute to different values, or change different attributes.

9. SPECIFICATION OF UCON 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 attributes. 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 obligations in UCON, we only focus on pre-conditions and ongoing-conditions. Since post-conditions do not affect the current usage request, we will consider this in an

administrative model in future work. Therefore there are only two core condition models in UCON, $preC_0$ and onC_0. Since a condition is built from the system attributes, the core models are defined to be immutable, i.e., the subject or object's attributes are not updated before, during, or after an access. Note that this is different from the fact that system attributes can be updated according to the environment, which is not captured by the UCON core models.

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

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

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

The policy of onC_0 is just:

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

As for $preC_0$, this policy is similar to that of onA_0 except for the condition predicates.

Example 10 Suppose that a day-shift user $dayshifter$ can access an object only during daytime. We define the system time $current\_time$ as an attribute, denoting an environment status, not an attribute of any subject or object. This is a combined model of $preC_0$ and onC_0. The policies can be expressed as:

$$(s.role = dayshifter) \wedge (8am \leq currentT \leq 5pm) \rightarrow permit(s, o, r)$$
$$tryaccess(s, o, r) \wedge permit(s, o, r) \rightarrow \bigcirc\big(permitaccess(s, o, r)\big)$$
$$permitaccess(s, o, r) \rightarrow \Box\big(\neg(8am \leq currentT \leq 5pm) \wedge$$
$$(state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$$

The first two rules are the same as those in $preC_0$, which specify that before an access, the subject's role must be $dayshifter$ and the system's time between 8am and 5pm. The third rule is for onC_0, where the whole accessing process must be taking place during daytime.

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 UCON model to show how to express some access control policies.

10.1 Role-based Access Control Models

In role-based access control (RBAC) [Sandhu et al. 1996], a role is a collection of permissions, and a permission is a pair (object, right). 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 a 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 11 Consider an RBAC1 model [Sandhu et al. 1996] where all the roles R are in a partial order hierarchy with respect to the dominate 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 values sets 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 usage control policy for RBAC1 is expressed by:

$$(role_1 \in s.actRole) \wedge ((role_2, r) \in o.perRole) \wedge (role_1 \geq role_2) \rightarrow permit(s, o, r)$$
$$tryaccess(s, o, r) \wedge permit(s, o, r) \rightarrow \bigcirc\big(permitaccess(s, o, r)\big)$$

This is a basic $preA_0$ policy, where the first rule specifies that if there is $role_1$ in the subject's $actRole$ attribute, $(role_2, r)$ is in the object's $perRole$ attribute, and $role_1$ dominates $role_2$ in the role hierarchy, 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.

Example 12 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 explicitly the user-role assignments. We can express this model in UCON $preA_1$ as follows:

$$(role_1 \in s.asgRole) \wedge (role_1 \notin s.actRole) \wedge (role_3 \notin s.actRole) \wedge ((role_2, r) \in o.perRole) \wedge (role_1 \geq role_2) \rightarrow permit(s, o, r)$$
$$permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge permit(s, o, r) \wedge \Diamond preupdate(s.actRole)$$
$$preupdate(s.actRole) : s.actRole' = s.actRole \cup \{role_1\}$$

The first 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), and there is a junior role $role_2$ of $role_1$ such that $(role_2, r)$ is in the object's $perRole$, 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 13 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. The Chinese Wall policy for $read$ is:

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

$$(o \notin s.ao) \wedge (o.class \notin s.ac) \rightarrow permit(s, o, read)$$
$$permitaccess(s, o, read) \rightarrow \blacklozenge\big(tryaccess(s, o, read) \wedge permit(s, o, read) \wedge$$
$$\Diamond preupdate(s.ac) \wedge \Diamond preupdate(s.ao)\big)$$
$$preupdate(s.ac) : s.ac' = s.ac \cup \{o.class\}$$
$$preupdate(s.ao) : s.ao' = s.ao \cup \{o\}$$

The first part of this policy is a $preA_0$ model, 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 part is a $preA_0$ model 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.

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 a history of subjects accessing this object. Here we present a simple example of object-based DSoD from [Simon and Zurko 1997].

Example 14 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 policy is specified below.

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

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

This is a basic $preA_1$ model. The first two rules say 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 last two rules specify 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 7. 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 15 Suppose L is a lattice of security labels with domination 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 of $classification$. All these attributes have a value domain of a lattice L. The authorization policy for $read$ is:

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

where LUB is the 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 obligations and conditions.

Example 16 Suppose that a medical doctor (s) can perform (r) a particular operation (o) only if he has operated more than 3 times before[7]. 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 policies are:

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

Example 17 In this example, a medical 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:

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

[7] These examples just show applications of TLA formulas as usage control policies, but does 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 $exp \leq 3$), i.e., with presence of senior doctors. This is not included here.

The left side of the first formula is a mix of an authorization predicate and an obligation, both of which must be true before the access can start.

Example 18 In this example, a junior medical doctor can perform an operation only when the monitoring system is running in the operating room. We model the running status of the monitoring system ($monitoring$) as a system attribute with value in $\{on, off\}$. This model is a combination of $preA_0$ and $preC_0$.

$$(s.role = junior\_doctor) \wedge (monitoring = on) \rightarrow permit(s, o, operate)$$
$$tryaccess(s, o, operate) \wedge permit(s, o, operate) \rightarrow \bigcirc\big(permitaccess(s, o, operate)\big)$$

11. SECURITY VERIFICATION

We have defined the logical models for UCON with temporal logic. A usage control policy is described by a logic formula that specifies the system state transitions by following the actions defined in a logic model. In general, a security policy is a static property in the form of a logical formula, based only on state predicates without actions, which has to be valid in every state. A system administrator or designer has to make sure that the security properties are satisfied. In this section we investigate the security property verification in our proposed logical UCON models. First we describe the process for the dynamic separation of duty policy in the model presented in Section 10.3, then we discuss the general security policy enforcements in general UCON systems using model checking.

11.1 An Example

In Section 10.3 a logic model is presented to enforce a DSoD property. The two usage control polices in this model specify the rules governing state changing in the system. DSoD security requires that, in each system state, a check's preparer is not the same as the issuer ($o.preparer \neq o.issuer$). This must be valid in each state of the system. Note that we adopt a strict equality here so that a $null$ value is not equal to another $null$ value by definition (e.g., in the initial state of the system).

For simplicity, we consider a system with a minimal set of entities to issue a check. According to the model presented in Section 10.3, the system has two subjects s_1 and s_2, and a check object o. Each subject has two attributes, sid and $role$, with $s_1.role = clerk$, $s_2.role = supervisor$, and $supervisor > clerk$. Both subject attributes are administrator-controlled attributes, and are not updated in the system by the usage control policies. Each object has two attributes, $preparer$ and $issuer$, whose value domain consists of all possible sids in the system. Specifically,

$$D_{preparer} = \{null, s_1, s_2\}$$
$$D_{issuer} = \{null, s_1, s_2\}$$

The two generic rights in the system are $prepare$ and $issue$, and both subjects can generate access requests on the object in this system. Therefore, there are only four possible usage state attributes:

—$state(s_1, o, prepare)$

—$state(s_1, o, issue)$

—$state(s_2, o, prepare)$

—$state(s_2, o, issue)$

Each of them can be assigned a value from the domain $\{initial, requesting, accessing, end, denied, revoked\}$. From Section 10.3 we know that the logic model specifying the system is a $preA_1$ model, in which only the three actions $tryaccess$, $permitaccess$, and $preupdate$ can occur. Therefore each of these state functions can have only the values $initial$, $requesting$, $accessing$, and end [8].

A particular system state s is defined by these subject and object attributes and their values. Since the subject attributes are fixed, a system state is determined by the two object attributes and the possible four usage state attributes, each of which can have one of four possible values. Therefore there are $3 \times 3 \times 4^4 = 2304$ different possible states of the system overall. From the initial state of the system, in which both $preparer$ and $issuer$ have value $null$ and all usage state attributes have value $initial$, most of these states cannot (fortunately) be reached in the system's life time, and can be ignored. For example, a non-initial $state(s_1, o, prepare)$ value cannot coexist with any other non-initial value of $state(s_2, o, prepare)$ since there is only one subject preparing the check at any time; similarly for $state(s_1, o, issue)$ and $state(s_2, o, issue)$. Actually we can see that s_1's role is $clerk$, which cannot issue a check, and therefore $state(s_1, o, issue)$ is not in any state of the system.

With these possible states and according to the usage control policies specified in Section 10.3, the state transitions of the system can be drawn as the directed graph shown in Figure 6. In this graph, t_0 is the initial state; an edge between states t_i and t_j indicates a valid action between these two states. For simplicity, the attributes with fixed value are not shown in the states except for the initial one.

As Figure 6 shows, there are two different paths to issue a check. In the left path, s_1 prepares the check, and after that only s_2 can issue it, since the $permit(s, o, issue)$ predicate requires that the issuer's sid be different from the object's $preparer$ attribute value. On the right path, s_2 prepares the check, but after this action, the check cannot be issued, since $s_1.role$ is $clerk$, which is not permitted to issue a check, while s_2 itself could not issue. Therefore no more actions can be performed after state t_{12} according to the usage control policies.

To check if the DSoD is enforced by the model, we check the predicate $o.preparer \neq o.issuer$ in each reachable state of the system from the initial state. Specifically, in Figure 6 all the states along both paths have to be checked.

11.2 General Security Analysis with Model Checking

For a general TLA model in which the system has a finite number of states, the security verification can always be checked using the mechanism introduced above. UCON core models do not include actions to create new subjects and objects and therefore, if the domain of each attribute is a finite set, the set of all possible states of the system is finite. In this section we introduce model checking with computation tree logic (CTL) to verify security properties in general UCON systems.

A CTL formula describes the property of a computation tree[9]. This tree shows all possible state changes starting from the initial state. The terms of CTL are state predicates, and a CTL formula consists of predicates which are connected with path quantifiers and

[8] It is assumed that $endaccess$ is an action in all pre-authorization models.

[9] Generally, starting from an initial state, each possible sequence of state transitions can be regarded as a path in a tree structure, of finite or infinite length, where the states may be repeated along a path.

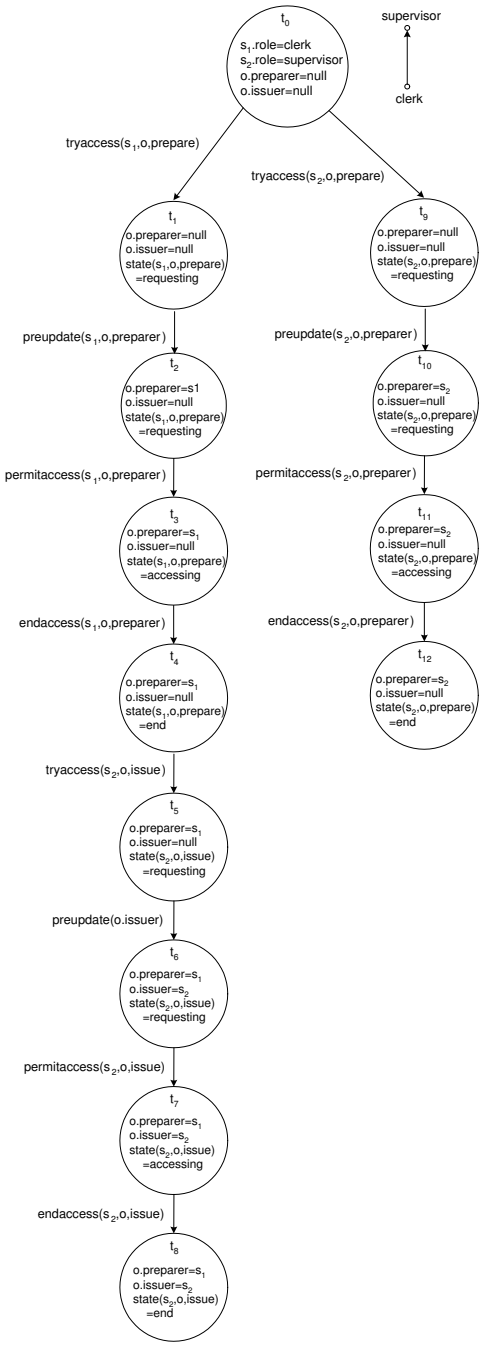

Fig. 6. State transitions in a DSoD system

temporal connectives. The two path quantifiers are A (for all paths) and E (there exists one path), and the general temporal connectives[10] are X (next state), F (some future state), G (all future states), and U (until). A CTL formula can be defined by the BNF:

$$\phi :\equiv \top|\bot|p|\neg\phi|\phi\wedge\phi|\phi\vee\phi|\phi\rightarrow\phi|AX\phi|EX\phi|A[\phi U\phi]|E[\phi U\phi]|AG\phi|EG\phi|AF\phi|EF\phi$$

where p is a state predicate, $\top :\equiv p\vee\neg p$, and $\bot :\equiv p\wedge\neg p$. For example, in a system state s, $EF\phi$ is true if there is a path starting from s, along which there exists at least one state s' in which the formula ϕ is true; $AG\phi$ is true if along every path from s, ϕ is true in every state.

The first step of model checking for security verification in UCON is to specify a static security policy or property in a state formula. For example, the security property of the DSoD policy in Section 10.3 is $o.preparer \neq o.issuer$. Since a security property must be true in every state of the system, the CTL form of the security policy is $AG\phi$. For example, the $AG(o.preparer \neq o.issuer)$ is the formula for the DSoD system in the previous section. As another example, for the usage control policies defined for the Chinese Wall policy in Section 10.2, the security policy is specified as

$$AG\neg\big((o_1 \in s.ao) \wedge (o_2 \in s.ao) \wedge (o_1 \in c) \wedge (o_2 \in c) \wedge (c \in s.ac)\big)$$

This formula states that in any state of the system, there are no two different objects in the same conflict class that have been accessed by a single subject.

For any state formula ϕ, we have $AG\phi :\equiv \neg EF\neg\phi$, and $EF\phi :\equiv E[\top U\phi]$, and therefore $AG\phi :\equiv \neg E[\top U\neg\phi]$. Thus, the verification of $AG\phi$ is equivalent to the verification of $\neg E[\top U\neg\phi]$. In general, the verification of $E[\phi_1 U\phi_2]$ can be performed by the following informal labelling algorithm:

—If ϕ_2 is true in a state, label it with $E[\phi_1 U\phi_2]$.

—If ϕ_1 is true in a state s, and at least one of its successors is labelled with $E[\phi_1 U\phi_2]$, label s with $E[\phi_1 U\phi_2]$.

—Repeat these two steps until there is no change.

For a state formula ϕ, a state is labelled with $AG\phi$ if it is not labelled with $E[\top U\neg\phi]$. A successful enforcement of a security policy requires that all reachable states from the initial state of the system be labelled with $AG\phi$. The complexity of this algorithm is a function of number of system states, which depends on the subject/object attributes and their value domains. Specifically, the number of the states in a system is exponential in the number of attributes and their domain sizes. There are some mechanism to overcome this "state explosion" problem in model verification, which is beyond the scope of this paper. If the number of subjects or objects of a system is unbounded, or the attribute value domains are infinite, the state of the system is infinite, and techniques of model checking for infinite state system may be applicable.

12. 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

[10]Note that the temporal connectives in CTL have different semantics from the temporal operators in our logic model. In CTL, a temporal connective's scope is all the system's states along a path, while in our logic model, a temporal operator's scope is the states in a single access process, which is a subset of the states along a path.

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. [Gal and Atluri 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. [Siewe et al. 2003] apply interval temporal logic to express and compose access control polices, and Hansen and Sharp [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 use TLA in our logic specification, and 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. [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. [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 [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.

13. CONCLUSIONS AND FUTURE WORK

We have developed a logic specification of UCON with temporal logic of actions. A logic model is given by a sequence 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 the 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 formulas represent usage control policies and are built from authorization predicates, actions, and system predicates. The

powerful specification capability and flexibility of the extended TLA strengthens UCON with precise modeling and specification. With the logically specified UCON system, we investigate the security verification with model checking mechanisms such as the labelling algorithm for finite state system.

This work opens several directions for further investigation. First of all, we need to develop administrative models for UCON, including attributes management, administrative policies, etc. UCON is attribute-based, and this requires synchronized attribute acquisition and management. Also 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 objectives 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.

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. We can apply tools such as TLA+ [Lamport 2003] to specify access control in such open and concurrent environments.

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).

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

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*. ACM.

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*. IEEE, Oakland, California.

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*. ACM.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May), also available from http://research.microsoft.com/users/lamport/tla/papers.html.

LAMPORT, L. 2003. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

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. 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), pp.38-47, 1996 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*. ACM.

SIMON, R. T. AND ZURKO, M. E. 1997. Separation of duty in role-based environments. In *IEEE Computer Security Foundations Workshop*. 183–194.