FORMAL MODEL AND ANALYSIS OF USAGE CONTROL

by

Xinwen Zhang
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of the
the Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	Ravi S. Sandhu, Dissertation Director
_____	Francesco Parisi-Presicce, Dissertation Co-director
_____	Larry Kerschberg
_____	Kris Gaj
_____	Daniel A. Menascé, Associate Dean for Research and Graduate Studies
_____	Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: _____ Summer Semester 2006
George Mason University
Fairfax, Virginia

Formal Model and Analysis of Usage Control

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Xinwen Zhang
Bachelor of Engineering
Huazhong University of Science and Technology, 1995
Master of Engineering
Huazhong University of Science and Technology, 1998

Director: Ravi S. Sandhu, Professor
Department of Information and Software Engineering
Co-director: Francesco Parisi-Presicce, Associate Professor
Department of Information and Software Engineering

Summer Semester 2006
George Mason University
Fairfax, Virginia

Dedication

To my parents, who have always inspired and encouraged me to continue my lifelong dream.

My special dedication goes to my lovely wife, Wei Xiong, who has been always supporting me and sharing all the difficult times with great love and sacrifices.

To my brothers and sisters, brothers-in-law and sisters-in-laws, who have given me great support for my study.

Acknowledgments

I would like to express my sincere appreciation and gratitude to my dissertation director, Professor Ravi Sandhu, who has enlightened and guided me throughout my doctoral studies. Great thanks to Dr. Sandhu who made this work possible, and encouraged me during my difficult times.

Special appreciation and thanks to my dissertation co-director, Professor Francesco Parisi-Presicce, who has guided me throughout my dissertation work and has been extending my knowledge and research skills, and discussing with me with great insights.

I am also grateful to my dissertation committee members, Professor Larry Kerschberg and Professor Kris Gaj for their valuable comments and suggestions.

My appreciation also goes to many friends at George Mason University for their help and to my co-authors for their collaboration.

Table of Contents

List of Tables

List of Figures

Abstract

FORMAL MODEL AND ANALYSIS OF USAGE CONTROL

Xinwen Zhang, Ph.D.

George Mason University, 2006

Dissertation Director: Ravi S. Sandhu

Dissertation Co-director: Francesco Parisi-Presicce

The concept of usage control (UCON) was introduced as a unified approach to capturing a number of extensions for access control models and systems. In UCON, a control decision is determined by three aspects: authorizations, obligations and conditions. Attribute mutability and decision continuity are two distinct characteristics which are presented in UCON for the first time. In this research I develop a logical model beyond the conceptual UCON model to capture the formal semantics of these key features, and then analyze the expressive power and safety properties of UCON.

Although the informal study of policy specification flexibility with UCON has been conducted in previous work, the multiple control components and unique features such as decision continuity and attribute mutability have not been formally studied. In this dissertation I develop a logical model of UCON based on an extended version of Lamport's temporal logical of actions (TLA) to formalize the state transitions in a single usage process. The model consists of predicates on subject and object attributes as authorizations, subject actions as obligations, and predicates on system attributes as conditions. With these basic terms, a usage control policy can be specified by a set of logical formulae, which are

instantiated from a fixed set of scheme rules. The policy specification language is shown to be sound and complete. The flexibility of policy specification with UCON is shown by expressing policies for various applications.

To formally study the expressive power of UCON by comparing with traditional access control models, a policy-based model is developed to formalize the overall effect of a usage process. With this model, I prove that the general single-object typed access matrix (SO-TAM) model can be simulated with a UCON $preA$ model, which is a sub-model of UCON with only pre-authorizations. The study of the expressive power shows that $preA$ is at least as expressive as the augmented typed access matrix model (ATAM). For the expressive power of UCON pre-obligation models ($preB$), I prove that a general UCON $preA$ model can be reduced to a $preB$ model, and vice versa. This demonstrates that fundamentally these two models have the same expressive power. For UCON ongoing authorization and obligation models (onA and onB), the system state changes non-deterministically, depending on concurrent accesses and reasons for attribute updates (e.g., ended access vs. revoked access). The study of the expressive power for these models is left for future work. In UCON pre-condition and ongoing condition models ($preC$ and onC), a usage control decision is determined by some environmental restrictions dependant on system attributes. Since UCON core models do not capture how system attributes change, it would be inappropriate to compare the expressive power of UCON condition models with others.

Safety is a fundamental problem of access control models. With the policy-based model, I first show that the general UCON $preA$ and $preB$ models have undecidable safety. With some restrictions on the general models, I propose a UCON $preA$ model with decidable safety. The restricted model maintains reasonable expressive power as shown by simulating a role-based access control (RBAC) model with a specific user-role administration scheme, and a digital rights management (DRM) application with consumable rights. The safety analysis of onA, $preB$, and onB is left for future work. For UCON condition models, since how system attributes change is not captured in UCON, the safety problem

is not a valid problem because the system state changes occur by events outside the scope of the control of UCON model.

Chapter 1: Introduction

1.1 Usage Control

Traditional access control models such as lattice-based access control (LBAC) [7, 18, 47] and role-based access control (RBAC) [19, 50] primarily consider static authorization decisions based on subjects' pre-assigned permissions on target objects. Access matrix models such as HRU [23] and TAM [46] use a matrix to distribute permissions at the discretion of individual subjects. In policy-based authorization management systems [11, 17, 25, 26], a centralized reference monitor (or distributed reference monitor with centralized administration) checks a subject's permission when access is requested, and the request is granted according to system security policies at the time of the access request. Once a subject is granted a permission, there are no further security checks for continued access.

Developments in information technology, especially in electronic commerce applications, require additional features for access control. On one side, in recent information systems, an access control decision can be determined by many aspects, such as general subject or object attributes, or some system constraints. For example, a professor can access the information of the students in his class only in his office and only during working hours. Further, an access may require some actions to be fulfilled by the subject, or by another subject instead of the requesting subject. For example, before reading an email, a user needs to send the acknowledgement of receipt to the sender. Traditional access control models cannot capture the multiple aspects of decisions in these applications. On the

other side, the usage of a digital object may be not only an instantaneous access or activity, like read and write, but also temporal and transient, such as payment-based online reading, metered by reading time or chapters, or a downloadable music file that can only be played 10 times. In these cases, a subject's permission may decrease, expire, or be revoked along with the usage of the object.

As traditional identity-based and role-based access control models cannot satisfy these purposes, UCON was recently proposed to be the next generation access control model that extends traditional access control models in multiple aspects [39] and fits new security requirements. In UCON, an access may be an instantaneous action, but may also be a process lasting for some duration with several related and subsequent actions. Actions and events during an access process may result in changes to the system state, such as subject or object attributes, or in changes in the status of an access (e.g., revoke an access). Usage control can be enforced before or during an access process, or both. A usage decision in UCON is made by policies of authorizations, obligations, and conditions (also referred as UCON_{ABC} core models). An authorization decision of an access is determined by the subject and/or object attributes. Obligations are actions that are required to be performed before or during an access process. Conditions are environment restrictions that are required to be valid before or during an access. An extreme example of UCON is the traditional access control models, in which the authorization decision is made instantly when an access request is generated, and there is no further check after that. By considering more general subject and object attributes, UCON is a comprehensive model to represent the underlying mechanism of existing access control models and policies. Beyond that, by combining authorizations, obligations, and conditions in access control decisions, UCON fundamentally extends the

traditional access control models and captures the access control requirements in DRM, trust management, and other modern information systems.

Two distinguishing features of UCON beyond traditional access control models are the continuity of access decision and the mutability of subject attributes and object attributes. In UCON, authorization decisions are not only checked and made before an access, but may be repeatedly checked during the access. A granted access may be revoked by the system if some policies are not satisfied, according to the changes of the subject or object attributes, or environmental conditions, or some obligations that are not fulfilled during the accessing.

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 has been changed as a side-effect of a previous access. This change, consequently, restricts the future 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 states of other accesses.

1.2 Expressive Power and Safety Analysis

The main goal of an access control model is to define and enforce security policies in a security system. Informally, the expressive power of an access control model is the capability of expressing various policies. As a fundamental problem, expressive power has been studied with traditional access control models since the introduction of access matrix model formalized by Harrison, Russo, and Ullman (HRU) [23]. Some related work on this aspect is summarized in Chapter 4. As UCON is claimed to fundamentally extend traditional access control models, a natural concern is its expressive power. With general usage-related subject and object attributes, UCON can be configured to support various policies for different applications. Also, the features of multi-aspect decision components, decision continuity, and attribute mutability greatly enhance its expressive power, as shown in Chapter 3. Another approach to study the expressive power of an access control model is to express another model by simulation, whereby the two models can be compared with regard to their relative expressive powers. The expressive power of UCON can be studied by simulating traditional access control models, such as access matrix models and RBAC.

A different but related and fundamental problem is the leakage of permissions in an access control model. In an access control system, a permission is granted or an access is authorized depending on the current state of the system. Also, the granting of a permission may consequently change the configuration of the system, and this, in turn, may enable

other permissions. Typically, a configuration of a system consists of a set of subjects, a set of objects, a set of rights, and a collection of assertions indicating whether a subject can have a right on an object. An access control system also contains a set of policies or rules to specify how the granting of a permission can change a system configuration or a state. For example, in an access matrix model, in each system state the matrix contains the rights that a subject has on an object, and there is a set of commands with which the matrix can be changed. For UCON, both the permission distribution and state change are determined by a set of policies. A policy refers to an access right that a subject can have on an object, based on attribute predicates, obligation actions, and system conditions. In a given system state, the permissions of a subject are evaluated by the attribute values, obligation satisfactions, and system status. As a side-effect of granting an access, one or more subject and/or object attributes can be changed, which result in a new system state. This dynamic property makes it difficult to foresee a system state in which a subject may have a particular right on a particular object. This is referred to as the safety problem in access control models.

The requirement of strong expressive power and that of a tractable safety property have been conflicting since the introduction of protection models in 1970's. It is not a surprising fact that for a given access control model, the more expressive power it has, the harder it is, computationally, to carry out safety analysis.

1.3 Problem Statement

Park and Sandhu [38, 39, 51] 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. Although the flexibility of policy specification has been informally studied in [39, 40, 60], to formally understand the concept of UCON, especially the comprehensive consideration of usage decisions, a formal specification of the principles of UCON and its flexible expressive capability is necessary. With a logical specification, we provide a tool to precisely define policies for system designers and administrators. With a conceptual and informal model, the capability to rigorously define policy is limited. Also, a logical specification provides the precise meaning of the new features of UCON, such as the mutability of attributes and the continuity of usage control decisions. Finally, to analyze general properties of UCON models such as expressive power and safety problem, we need a formal model.

For an access control model, expressive power and safety analysis are two fundamental problems. Previous work [39, 40] has informally shown the expressive power of UCON, while a formal study on this aspect remains to be done. Generally, the expressive power of an access control model can be evaluated by comparing it with other models, i.e., by simulating one model with another model. In particular, as UCON is claimed to be a flexible and comprehensive model, we need to understand its relative expressive power with respect to traditional access control models.

The safety problem of an access control system is to determine if a subject can get a permission on an object in some reachable state of the system. Since UCON is shown to have strong flexibility and expressive power for policy specification, it is a reasonable

conjecture, although needs to be proven, that the safety problem of UCON is undecidable in general. How to introduce reasonable constraints on the general model and obtain a decidable model with practical expressive power is a fundamental problem for UCON.

1.4 Summary of Contributions

This dissertation contains the following contributions.

- A logical model of UCON is developed with TLA to formalize the state transitions in a single usage process. With this model,
  - policy specifications for core UCON models are presented; and
  - a fixed set of scheme rules are defined to specify general UCON policies with the properties of soundness and completeness; and
  - policy specification flexibility of UCON is illustrated by expressing various policies.
- Policy-based formal models of UCON $preA$ and $preB$ are developed to formalize the accumulative effect of a usage process. The expressive power of $preA$ and $preB$ is studied and the following results are achieved.
  - By simulating SO-TAM with $preA$, $preA$ is proved to be at least as expressive as SO-TAM.
  - Further, $preA$ is shown to be more expressive than SO-TAM and TAM, and at least as expressive as ATAM.
  - The $preA$ and $preB$ have the same expressive power.

- The safety property of UCON is analyzed with the policy-based model and the following results are achieved.
  - The safety problem of the general $preA$ and $preB$ is undecidable.
  - The safety problem is decidable for a $preA$ model with finite attribute domains and without creating policies, and the problem is polynomial in the number of possible states of the system and NP-hard in the number policies in the scheme.
  - The safety problem is decidable for a $preA$ model with finite attribute domains and creating policies, and the attribute creation graph is acyclic and there are no cycles that include create-parent tuple in attribute update graph.
  - The decidable $preA$ models maintain practically useful expressive power as shown by specifying an RBAC model with a user-role assignment administrative scheme and a DRM application with consumable rights.

1.5 Organization of the Dissertation

Chapter 2 first introduces the OM-AM framework of security engineering as the background of this dissertation, and then the conceptual UCON model and a motivating example. In Chapter 3 a logical model is developed to formalize the state transitions in a single usage process, and its flexibility of policy specification is presented. In Chapter 4 a policy-based formal model focusing on the overall effect of a usage is developed and the expressive power of UCON $preA$ and $preB$ is studied. With this policy-based model, the safety problem of UCON $preA$ is studied in Chapter 5. Chapter 6 summarizes this dissertation and presents some directions for future work.

Chapter 2: Background

This chapter presents some background knowledge and work relevant to this dissertation. The general OM-AM framework for security engineering is introduced in the context of UCON, followed by the conceptual UCON_{ABC} model with its new features. The related work regarding temporal characteristics, expressive power, and safety analysis of access control models are presented in Chapter 3, 4, and 5, respectively.

2.1 OM-AM Framework

OM-AM framework is a layered approach to security system first proposed in the context of role-based access control (RBAC) [48] systems. As shown in Figure 2.1, the four layers are Objectives, Models, Architectures, and Mechanisms, surrounded by a basic requirement of assurance which permeates all layers. The objective layer captures the informal specifications of a system's security requirements (policies or goals). The model layer provides the abstract or formal interpretation of the security requirements. The architecture layer describes the security design and implementation strategy in terms of components, servers, brokers, etc., and their relationships. The mechanism layers focuses on concrete implementation techniques. In a high-level view, the objectives and models are concerned with articulating *what* the security goals and expressions are, and what should be achieved, while the architectures and mechanisms address *how* to meet these requirements. OM-AM

framework is neither a top-down waterfall-style nor process-based layers (e.g., software engineering process). Each layer's mapping to adjacent layers is many-to-many, e.g., a model can be supported by multiple architectures, while an architecture can support multiple security models. At the same time, each layer deals with distinct and independent functions, and these functions are tightly related to other layers to some degree.

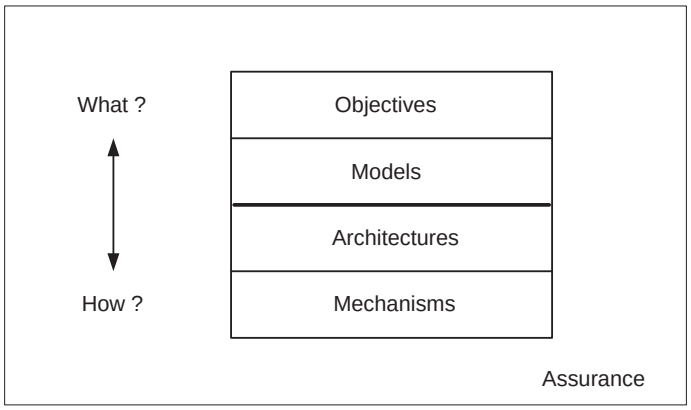

Figure 2.1: The OM-AM framework for security engineering

For the instance of RBAC systems shown in Figure 2.2, the objective layer is policy neutral since RBAC can be configured to express various policies [37]. In the model layer, there are many RBAC models with different features. Among them, RBAC96 [50] is the first comprehensive and well-accepted model, and ARBAC97 is an administrative model for RBAC systems. At the architecture layer, RBAC can be supported with server-pull, user-pull, or hybrid architectures [48]. At the implementation layer, there are many mechanisms that can be used, such as secure cookies [43], digital certificates [41], security assertion markup language (SAML) [35], SSL, IPSec, X.509, etc.

This dissertation focuses on the model layer of UCON. To some extent, the objective

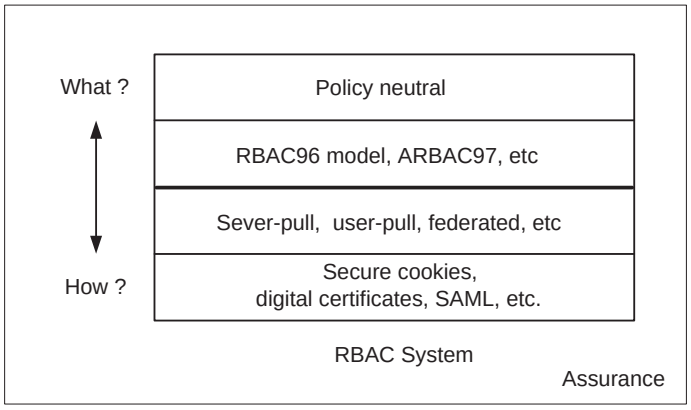

Figure 2.2: The OM-AM framework for RBAC systems

and architecture layers are also involved. As shown in Figure 2.3, at the objective layer, as UCON model is attribute-based, general subject and object attributes can be defined to support various security policies for different application requirements. This makes UCON policy neutral. In the model layer, the conceptual UCON_{ABC} model has been previously proposed [39] and a formal model is presented in this dissertation. Decision components such as authorizations, obligations, and conditions are integrated in a single model, and can be configured to express traditional access control models and various security policies such as separation of duty, Chinese Wall, etc. [39, 59]. In the architecture layer, traditional server-side reference monitor (SRM) or emerging client-side reference monitor (CRM) or combination of them can be used to support a UCON system [42]. For the implementation mechanism, existing DRM technologies, such digital watermarking, can be used in some applications. At the same time, emerging trusted computing (TC) [1–3] technologies provide mechanisms to support client-side reference monitors to enforce UCON policies [52]. For client-side policy enforcement, remote attestation for platforms and content viewers is

needed. Trusted computing technologies with the support of public key infrastructure (PKI) can be the concrete mechanisms for implementation. A UCON policy can be specified by XML with some standard approaches such as extensible access control markup language (XACML) [36] for security policies and extensible rights markup language (XrML) [58] for DRM policies.

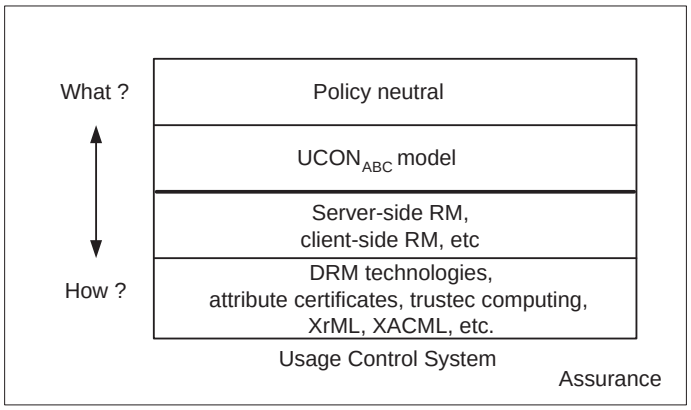

Figure 2.3: The OM-AM framework for UCON systems

2.2 UCON$_{\mathrm{ABC}}$ Model

As depicted in Figure 2.4, a usage control system has six components: subjects and their attributes, objects and their attributes, rights, authorizations, obligations, and conditions. The authorizations, obligations and conditions are components of usage control decisions. An authorization decision is based on the subject's and/or object's attributes. Obligations are activities that have to be performed by a subject before or during an access. Conditions are system environment restrictions which are required before or during an access.

The most important properties that distinguish UCON from traditional access control

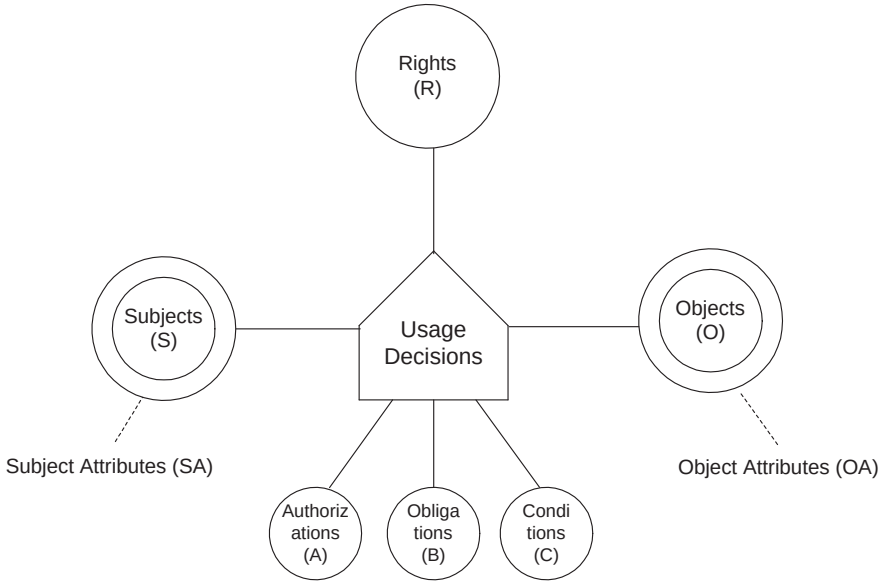

Figure 2.4: Usage control model

models and trust management systems are the continuity of usage decisions and the mutability of attributes. Continuity means that a usage control decision can be determined and enforced not only before an access, but also during the ongoing period of the access. Figure 2.5 shows a complete usage process consisting of three phases along the time line: before usage, ongoing usage, and after usage. Usage control decisions can be checked and enforced in the first two phases, named pre-decisions and ongoing-decisions, respectively [1].

[1] In the after usage phase, no decisions are checked and enforced since there is no access control after a subject finishes usage on an object. There can be obligations and conditions defined in this phase, which are called post-obligations and post-conditions, respectively. Since UCON is defined as a session-based access control model targeting the current access request and ongoing usage, post-obligations and post-conditions are not included in the core UCON model, but should be included in related administrative models. In this work we only focus on the core aspects of UCON, while an administrative model should be developed in the future.

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

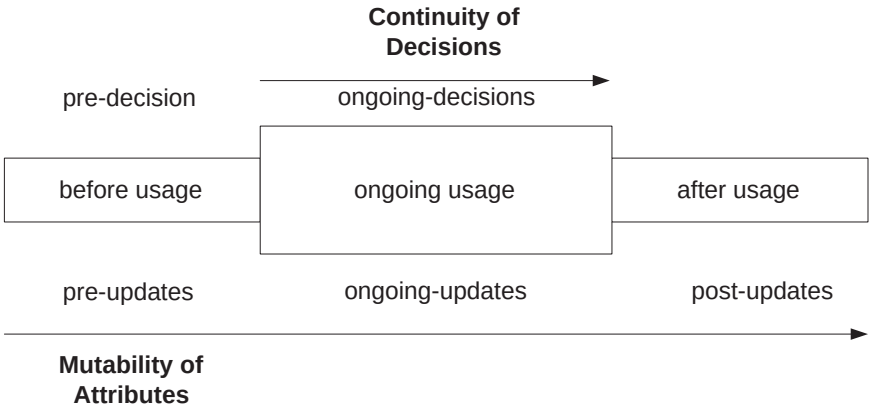

Figure 2.5: Continuity and mutability properties of UCON

2.2.1 Core Models

For each decision component (authorizations, obligations, and conditions) in UCON, several core models are defined based on the phase where usage control is checked and updates are performed. For example, in authorization core models, the usage control decisions are dependent on the subject and object attributes, which can be checked and determined in the

first two phases of an access. Based on the possible updates in all three phases, eight core authorization models can be defined as follows.

- $preA_0$: a usage control decision is determined by authorizations before the usage, and there is no attribute update before, during, or after this usage.
- $preA_1$: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated before this usage.
- $preA_2$: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated during this usage.
- $preA_3$: a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated after this usage.
- onA_0: usage control is checked and the decision is determined by authorizations during the usage, and there is no attribute update before, during, or after this usage.
- onA_1: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated before this usage.
- onA_2: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated during this usage.
- onA_3: usage control is checked and the decision is determined by authorizations during the usage, and one or more subject or object attributes are updated after this usage.

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

Similar obligation and condition core models can be defined. A real model is typically a combination of multiple core models. Figure 2.6(a) shows all possible combination of core models and their relationships, where the $\mathrm{UCON_A}$, $\mathrm{UCON_B}$, and $\mathrm{UCON_C}$ are the three base models at the bottom. Any two of them can be combined to form a new model, and all together form the $\mathrm{UCON_{ABC}}$ model. Each of the A, B, and C models is divided into several cases as shown in Figure 2.6(b), (c), and (d), respectively. In each of them, the mutable cases of pre-update model (1), ongoing update model (2), and post-update model (3) dominate the immutable model (0), while there is no ordering among the mutable cases.[2]

[2]In [38, 39], the $preA_2$, $preB_2$, and all mutable condition core models ($preC_1$, $preC_2$, $preC_3$, onC_1, onC_2, and onC_3) are not included in the UCON core models. For $preA_2$ and $preB_2$, the reason was that since a decision is made before the usage, ongoing updates can be postponed to the after-usage phase in a usage process. As in a system there may exist concurrent usage processes, ongoing updates of an usage can affect other usages. Therefore these two core models are included here. For mutable condition core models, subject and/or object attributes can also be updated, such as usage time and usage log. Similarly, an update in a usage process of condition core models can affect other usage processes. Therefore, the mutable condition core models are also included in this work.

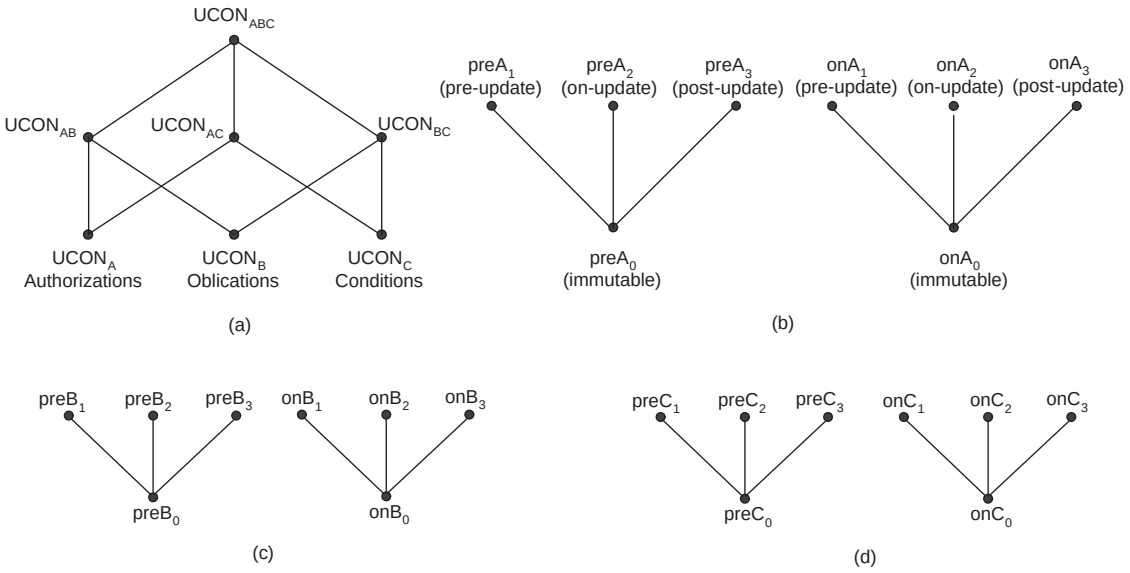

Figure 2.6: $\mathrm{UCON}_{\mathrm{ABC}}$ family of core models

2.2.2 Attribute Management and Mutability

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

Administrator-controlled attributes can be modified only by explicit administrative actions. These attributes are modified at the administrator's discretion but are "immutable" in that the system does not modify them automatically, unlike mutable attributes. Here the administrator can be either a security officer or a user, although in general, administrative

actions are made by security officers. Administrator-controlled attributes are typical in traditional access control policies such as mandatory access control (MAC) and RBAC. Static separation of duty and user-role assignment in RBAC are other examples of this case.

Unlike administrator-controlled, in system-controlled attribute management, updates are the side effects or results of the 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 administrator-controlled management the update involves administrative decisions and actions. This is why system-controlled attributes are "mutable" attributes that do not require any administrative action for updates. Attribute mutability is considered as part of UCON core models. In this dissertation 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 [40], including exclusiveness, accounting, immediate access revocation, obligations, and dynamic confinements.

2.2.3 An Example

In this section, an example motivating the new features of UCON is presented. Traditional access control models and policies have difficulties, or lack the flexibility to specify policies in these scenarios.

Consider a DRM application with a 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 possible policies to determine which user's ongoing access must be revoked. Among them,

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

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

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

For these three different policies [3], we need to define different attributes for subjects and objects, respectively.

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

(b) An object has the same attribute as in (a). Each subject has two attributes: the status of the subject with a value $busy$ or $idle$, and the continuous idle time in a single usage process. In order to monitor the idle time, the system has to check the status

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

and update the idle time during the entire ongoing access by means of ongoing-update. Similar to (a), there are pre-update, revoking access, and post-update actions. Revocation is performed with respect to the longest idle access when the total count of ongoing accessing subjects is larger than 10.

(c) Here again an object has the same attribute as in (a). Each subject has an attribute of accumulating usage time to record the total usage time of this subject on this object over the subject and object life. Similar to (a) and (b), there are pre-update, revoking access, and post-update actions. Revocation is performed with respect to the subject with the longest usage time access when the total count of ongoing accessing subjects is larger than 10. In addition, there is a post-update 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 and an object, but also with the system and other related processes which are accessing or trying to access the same object concurrently. An access decision is no longer a single function of (subject, object, right), but depends on the attributes of the entities (subject and object) 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 sequence of actions and events not only from a subject, but also from the system. This example also shows that a real system is the combination of several core models.

Chapter 3: Formal Model and Policy Specification

This chapter first reviews TLA and then defines some extensions to include past temporal operators. A logical model is developed to formalize the state transitions in a single usage process with this extended TLA. Policy specifications for UCON core models are presented. A set of scheme rules are proposed to specify general UCON policies and shown to have the properties of soundness and completeness. The flexibility of the policy specification language is illustrated at the end of this chapter.

3.1 Temporal Logic of Actions

Extending temporal logic [33] by introducing boolean valued actions, the temporal logic of actions (TLA) [30] is a powerful tool to specify systems and their properties, especially for interactive and concurrent systems. This section gives a brief introduction to the basic terms and the syntax of temporal formulae, and then introduce some additional temporal operators along with their semantics.

3.1.1 Building Blocks

Variables, values, and states are basic concepts in TLA. Values are elements of a data type. A variable has a name like x and y, and can be assigned a value. We assume that there is an infinite set of available variables with names x, y, etc., to which values can be assigned.

A constant is a variable that is assigned with a fixed value. A state s is characterized by assignment of a value $s[\![x]\!]$ to each variable x.

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

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

where $f(\forall \text{`}v\text{'}\colon s[\![v]\!]/v)$ is the value obtained by substituting $s[\![v]\!]$ for each variable v in the expression.

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

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

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

Since a predicate P is a boolean expression built from variables and constants, it is

regarded as a special action without primed variables. A pair (s, t) is a P step iff $s[\![P]\!]$ is $true$.

3.1.2 Temporal Formula and Semantics

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

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

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

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,

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

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

3.1.3 Extension of TLA

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

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

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

The *"Next"* and *"Until"* Temporal Operator

For a behavior $< s_0, s_1, s_2, ... >$, the semantics of the *Next* operator ($\bigcirc$) is defined as

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

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

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

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

There is an equivalence between these temporal operators:

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

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

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

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

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

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

$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![\ominus F]\!] \equiv s_{-1} [\![F]\!] s_0$$

$$< ..., s_{-2}, s_{-1}, s_0, s_1, s_2, ... > [\![F \mathcal{S} G]\!] \equiv$$

$$\exists i < 0 : \big(s_i [\![G]\!] s_{i+1} \wedge (i < j < 0 \rightarrow s_j [\![F]\!] s_{j+1})\big)$$

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

$$\blacklozenge F \equiv \neg \blacksquare \neg F$$

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

3.2 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.

3.2.1 Attributes and States

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

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

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

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

goes to an end state when a subject finishes the usage.

In UCON, a function is an expression built from one or more attributes and constants. Formally, a function is a mapping from a set of attribute values to a new value. For instance, in the example in Section 2.2.3, the total number of ongoing accessing subjects for an object is a function of the object's attribute (a list of accessing subjects).

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

3.2.2 Predicates

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

or from the system. A predicate can be defined with a number of attributes from a single entity, or two entities, or the system.

3.2.3 Actions

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

Usage Control Actions

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

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

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

If the system performs an update successfully, the attribute value is changed to a new value, and the action is $true$, otherwise, it is $false$. Note that in our model we do not consider the time delay of an action, and we assume that an action is always performed instantly causing the transition to the next state[1].

[1]In a real system, an update may be delayed or failed, or an update is performed but the target object is

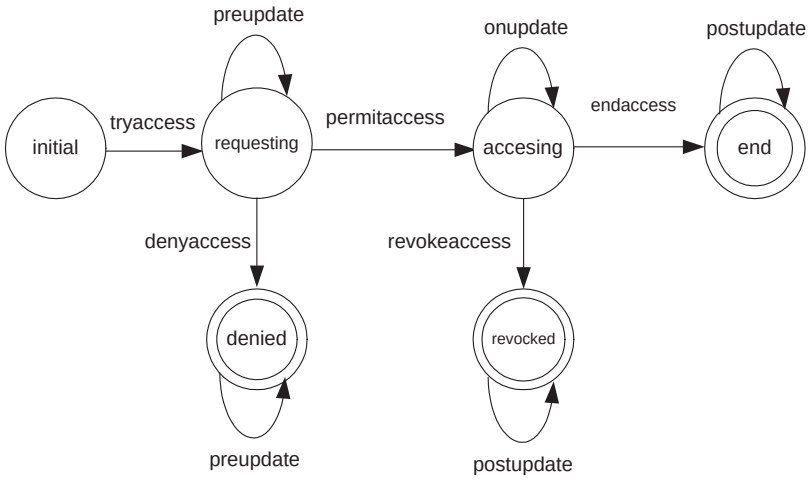

Figure 3.1: State transition of a single access with usage control actions

Another type of usage control action is performed by a subject or the system that change the status of an access (s, o, r). As mentioned before, there are six different possible values of $state(s, o, r)$ during an access life cycle. The transition from a state to another state is a usage control action, as shown in Figure 3.1. Note this figure only shows the changes of the usage status $state(s, o, r)$ in one usage process.

We can categorize all usage control actions into two classes: actions performed by a subject and actions performed by the system. Figure 3.2 shows these actions during the life cycle of a usage process. They are briefly explained below.

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

2. $permitaccess(s, o, r)$: grants the access request of (s, o, r), performed by the system.

not available, e.g., because of network problem or storage problem. Therefore there needs to be logging and recovering mechanisms to monitor the process. As standard approaches can be used for these purposes, the model does not capture these aspects for the sake of simplicity.

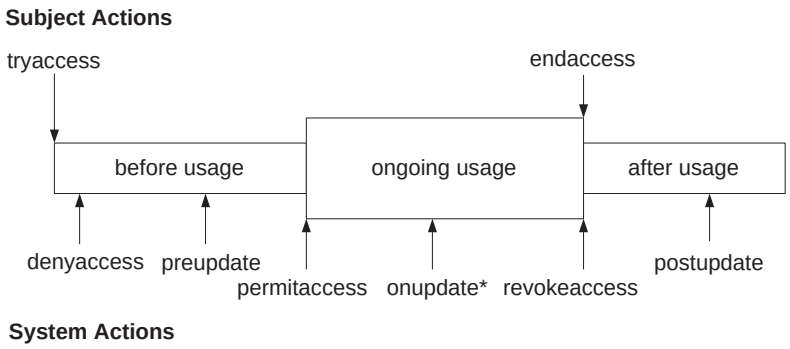

Figure 3.2: Usage control actions

3. $denyaccess(s, o, r)$: rejects the access request of (s, o, r), performed by the system.

4. $revokeaccess(s, o, r)$: revokes an ongoing access (s, o, r), performed by the system.

5. $endaccess(s, o, r)$: ends an access (s, o, r), performed by subject s.

6. $preupdate(attribute)$: updates a subject or an object attribute before granting access or after denying an access, performed by the system.

7. $onupdate(attribute)$: updates a subject or an object attribute during the usage phase, performed by the system.

8. $postupdate(attribute)$: updates a subject or an object attribute after access, performed by the system.

For a usage process, there can be multiple $preupdates$, $onupdate$, and $postupdate$ actions for different attributes before, during, and after the access, respectively. Also, in the ongoing usage phase there may be continual or periodical $onupdate$ actions for a single attribute, as the star symbol indicates in Figure 3.2.

Obligation Actions

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

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

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

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

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

3.2.4 Model and Satisfaction of Formulae

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

Definition 1. *A logical model of UCON is a 5-tuple:* $\mathcal{M} = (\mathcal{S}, \mathcal{P}_{\mathcal{A}}, \mathcal{P}_{\mathcal{C}}, \mathcal{A}_{\mathcal{A}}, \mathcal{A}_{\mathcal{B}})$*, where*

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

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

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

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

Definition 2. *A logical formula in UCON is defined by the following grammar in BNF:*

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

where a is an action, p is a predicate.

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

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

3.3 Specification of Authorization Core Models

For each decision component in UCON, several core models are defined based on the phase where updates are performed. This section presents the policy specifications of UCON authorization core models. Obligation and condition models are illustrated in the next two sections.

In authorization core models, usage control decisions are dependent on the subject and object attributes, which can be checked and determined in the first two phases of an access. Based on the possible updates in all three phases, eight core authorization models can be defined as shown in Section 2.2.1.

3.3.1 The Model $preA_0$

As presented in [39, 51], most traditional access control models can be expressed as $preA_0$ model, in which an authorization decision is determined by the system before the access happens, and there is no update for subject or object attributes. The usage control policy is:

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

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

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

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

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

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

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

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

granted, as the following formula states.

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

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

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

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

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

1. $permitaccess(s, o, read) \rightarrow$

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

2. $permitaccess(s, o, write) \rightarrow$

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

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

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

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

Besides MAC and DAC, many other examples of $preA_0$ policies can be similarly specified. Section 3.7 shows some of them.

3.3.2 The Model $preA_1$

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

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

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

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

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

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

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

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

According to the assumption mentioned in Section 3.3.1, the time line is bounded during the local usage process, so in this policy the "Once" operator does not refer to any past state before $tryaccess$ in a single usage process. Therefore in $preA_1$, the $preupdate$ action is performed after the $tryaccess$, the authorization predicates are required to be $true$ before the $preupdate$, and there is no constraint after the update action.

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

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

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

3.3.3 The Model $preA_2$

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

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

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

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

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

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

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

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

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

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

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

3.3.4 The Model $preA_3$

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

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

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

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

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

1. $permitaccess(s,o,read) \rightarrow$

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

2. $endaccess(s,o,read) \rightarrow \Diamond postupdate(s.expense)$

   $postupdate(s.expense) : s.expense' = s.expense + o.readingCost$

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

3.3.5 The Model onA_0

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

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

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

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

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

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

Since we are specifying the core aspects of UCON, pre-authorization rules are not included in ongoing-authorization models, and for simplicity the $tryaccess$ action implied by the $permitaccess$ action is ignored. The same holds for other ongoing core models. In practice, an application may require a combination of several core models. We discuss this in Section 3.6.

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

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

3.3.6 The Model onA_1

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

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

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

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

3.3.7 The Model onA_2

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

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

2. $permitaccess(s, o, r) \rightarrow$

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

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

$permitaccess(s, o, r) \rightarrow$

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

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

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

or, more generally,

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

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

3.3.8 The Model onA_3

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

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

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

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

In many applications, the update after an access ended by a subject, is different from the one after an access is revoked by the system, as shown in the second and third rules. Here

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

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

Example 6 Consider the usage control policies for the example in Section 2.2.3. In this example, an object attribute is a set of accessing subjects $accessingS = \{s | state(s, o, r) = accessing\}$. We also define the system $clock$ as a system attribute. For the different policies we define different subject attributes.

(a) *Revocation by the earliest start time*

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

1. $permitaccess(s, o, r) \rightarrow$

   $\blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(o.accessingS) \wedge \blacklozenge preupdate(s.startTime)$

   $preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$

   $preupdate(s.startTime) : s.startTime' = sys.clock$

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

3. $endaccess(s, o, r) \vee revokeaccess(s, o, r) \rightarrow \Diamond postupdate(o.accessingS) \wedge \Diamond postupdate(s.startTime)$

$postUpdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$

$postUpdate(s.startTime) : s.startTime' = null$

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

(b) *Revocation by the longest idle time*

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

1. $permitaccess(s, o, r) \rightarrow$

   $\blacklozenge tryaccess(s, o, r) \wedge \blacklozenge preupdate(o.accessingS) \wedge \blacklozenge preupdate(s.idleTime)$

   $preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$

   $preupdate(s.idleTime) : s.idleTime' = 0$

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

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

   $onupdate(s.idleTime) : s.idleTime' = s.idleTime + 1$

4. $endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow \Diamond postupdate(o.accessingS)$

   $postupdate(o.accessingS) : o.accessingS' = o.accessingS - \{s\}$

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

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

We define the accumulating usage time $usageTime$ as a subject attribute. The control policy is a combination of onA_1 and onA_3 as follows.

1. $permitaccess(s,o,r) \rightarrow$

   $\blacklozenge tryaccess(s,o,r) \wedge \blacklozenge preupdate(o.accessingS)$

   $preupdate(o.accessingS) : o.accessingS' = o.accessingS \cup \{s\}$

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

3. $endaccess(s,o,r) \vee revokeaccess(s,o,r) \rightarrow$

   $\Diamond postupdate(s.usageTime) \wedge \Diamond postupdate(o.accesingS)$

$$postupdate(o.accesingS) : o.accessingS' = o.accessingS - \{s\}$$

$$postupdate(s.usageTime) : s.usageTimes' = s.usageTime + sys.periodT$$

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

3.4 Specification of Obligation Core Models

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

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

1. pre-obligations: obligations that must have been performed before a subject starts to access an object.

2. ongoing-obligations: obligations that must be performed during a usage process.

Obligations that have to be performed after an access, since they only affect the future usage process, are considered as global obligations [39, 51]. For example, an action of a user clicking an agreement button before playing a music file is regarded as an obligation, while the payment action of a monthly billing is a global obligation, because this action does not affect the current usage access. In UCON an administration model is needed to capture global obligations. In this work, we only focus on the session-based usage control model, in which only obligations before and during the usage process are considered. The global obligations will be described in our future work.

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

- $preB_0$: a usage control decision is determined by obligations before the access, and there is no attribute update before, during, or after the usage.
- $preB_1$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated before the usage.
- $preB_2$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated during the usage.
- $preB_3$: a usage control decision is determined by obligations before the access, and one or more subject or object attributes are updated after the usage.
- onB_0: usage control is checked and the decision is determined by obligations during the access, and there is no attribute update before, during, or after the usage.

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

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

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

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

3.4.1 The model $preB_0$

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

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

where $ob_1, \ldots, ob_i$ are obligation actions for access (s, o, r). This rule requires that an access can be granted only after all the obligations are satisfied. The difference between $preB_0$ and $preA_0$ is that, in $preB_0$, all the obligations are satisfied before an access requested is granted, and generally may not be performed in the same state, so that the "Once"

operator is applied for each of them in the policy formula. In $preA_0$, instead, the authorization predicates are checked in a single state. As mentioned in Section 3.3.2, the "Once" operator does not refer to any state before the $tryaccess$ action in a single access process. This indicates that all obligation actions are for the current access request.

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

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

1. $permitaccess(s, o, order) \rightarrow$

   $\blacklozenge tryaccess(s, o, order) \wedge \blacklozenge click\_agreement(s, agree\_statement)$ □

3.4.2 The Model $preB_1$

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

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

   $\blacklozenge preupdate(attribute)$

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

3.4.3 The Model $preB_2$

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

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

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

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

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

or

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

or, more generally,

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

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

3.4.4 The Model $preB_3$

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

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

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

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

Example 8 In the Example in Section 3.4.1, a customer's $orderList$ is updated by adding the ordered item after he/she places an order. This can be expressed with a $preB_3$ policy as the following.

1. $permitaccess(s, o, order) \rightarrow$

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

2. $endaccess(s, o, order) \rightarrow \Diamond postupdate(s.orderList)$

   $postupdate(s.orderList) : s.orderList' = s.orderList \cup \{o\}$ □

3.4.5 The Model onB_0

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

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

   $revokeaccess(s, o, r)\big)$

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

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

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

or

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

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

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

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

3.4.6 The Model onB_1

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

1. $\Box\big(\neg(\bigwedge_i (p_{i1} \wedge \cdots \wedge p_{ik_i} \rightarrow ob_i)) \wedge (state(s,o,r) = accessing) \rightarrow revokeaccess(s,o,r)\big)$
2. $permitaccess(s,o,r) \rightarrow \blacklozenge tryaccess(s,o,r) \wedge \blacklozenge preupdate(attribute)$

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

3.4.7 The Model onB_2

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

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

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

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

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

or

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

or, more generally,

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

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

3.4.8 The Model onB_3

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

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

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

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

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

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

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

2. $permitaccess(s, o, r) \rightarrow \blacklozenge preupdate(s.UsageTime)$

   $preupdate(s.UsageTime) : s.UsageTime' = 0$

3. $\Box\big((state(s, o, r) = accessing) \rightarrow onupdate(s.UsageTime)\big)$

   $onupdate(s.UsagtTime) : s.UsageTime' = s.UsageTime + 1$

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

   $postupdate(s.UsageTime) : s.UsageTime' = 0$

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

3.5 Specification of Condition Core Models

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

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

1. pre-conditions: conditions that must be true before an access.

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

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

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

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

where pc_1, ..., pc_i are condition predicates built from system attributes. This policy is very similar to that of $preA_0$ and $preB_0$, except that the decision is determined by predicates of

system attributes, instead of the subject's and object's attributes in $preA_0$, and obligation actions in $preB_0$.

The policy of onC_0 is:

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

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

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

1. $permitaccess(s,o,r) \rightarrow$

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

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

   $revokeaccess(s,o,r)\big)$ □

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

3.6 Formal Specification of General UCON Models

After specifying the core models in UCON, we study the formal semantics of a general UCON model in this section. Specifically, we show that a general UCON policy can be expressed with a set of logical formulae instantiated from a fixed set of scheme rules, and a

set of logical formulae instantiated from these rules can be satisfied by at least one UCON model. These two properties are regarded as the completeness and soundness of our policy specification language.

3.6.1 Scheme Rules

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

For an access (s, o, r), let $pa_1, \ldots, pa_i$ be a set of authorization predicates, $ob_1, \ldots, ob_j$ be a set of obligation actions, and $pc_1, \ldots, pc_k$ be a set of condition predicates. According to the specifications of the core models explained in previous sections, a UCON policy can be specified by two kinds of logical rules: a usage control decision rule and an update rule. The following control rules (CRs) are specified for the pre-decision and ongoing decision of a single usage process, respectively.

$CR1$: $permitaccess(s, o, r) \rightarrow$

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

$CR2$: $\Box\big(\neg\big((\bigwedge_{n_i} pa_{n_i}) \wedge (\bigwedge_{n_j} (pb_{n_j 1} \wedge \cdots \wedge pb_{n_j k_{n_j}} \rightarrow ob_{n_j})) \wedge (\bigwedge_{n_k} pc_{n_k})\big) \wedge$

$$(state(s, o, r) = accessing) \rightarrow revokeaccess(s, o, r)\big)$$

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

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

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

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

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

$\Diamond(endaccess(s, o, r) \vee revokeaccess(s, o, r))\big)$

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

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

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

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

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

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

be a combination of multiple core models, which are specified by a set of the control rules and update rules.

3.6.2 Completeness and Soundness

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

Theorem 1. *(Completeness) Any UCON policy can be specified by a non-empty set of control rules and a set of update rules, each of which is instantiated from a unique scheme rule.*

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

By completeness we mean that a general UCON model introduced in Section 2.2.1 and conceptually defined in [38, 39] can be formally defined with our logical model. That is,

the set of scheme rules is adequate to specify policies for all UCON core models and any combination of them.

Theorem 2. *(Soundness) For a non-empty set of control rules and a set of update rules, each of which is instantiated from a unique scheme rule, there is at least one UCON model in which the system state transitions satisfy these rules.*

Proof. We construct a UCON model to satisfy eight logical formulae for a single access (s, o, r), one for a unique scheme rule. Consider the two control rules $CR1'$ and $CR2'$, which are instantiations of $CR1$ and $CR2$, respectively, and six update rules, $UR1'$, ..., $UR6'$, one for each unique scheme update rule, respectively. Without loss of generality, we assume that all the attributes in these update rules are different, since, as we have mentioned in Section 3.3, multiple updates on the same attribute can be reduced to a single update. Consider a system where a state is specified by the attributes (subject's, object's, and the system's) in all of the rules. Initially the system state is s_0, and $state(s, o, r) = initial$. The state transitions are constructed with the following steps and illustrated in Figure 3.3.

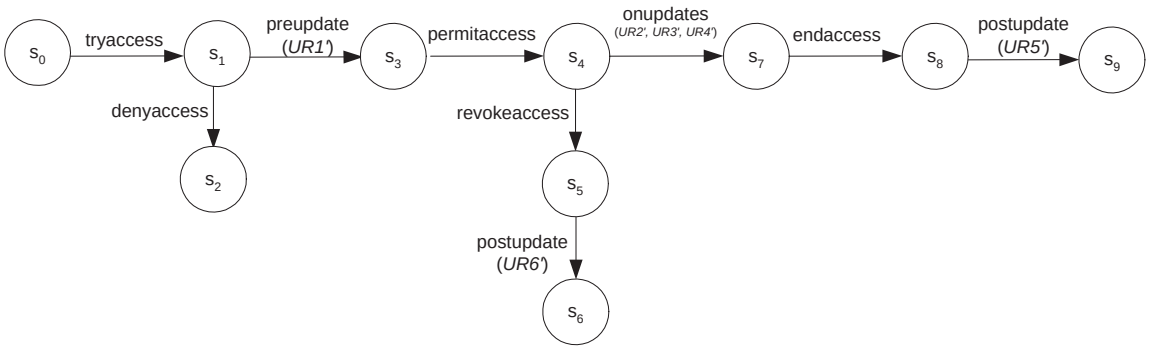

Figure 3.3: State transitions

- In s_0, the subject s generates an access request ($tryaccess$) to o with right r, the value

of $state(s, o, r)$ is changed to $requesting$, and the system's new state is s_1. The other attributes have the same values as in s_0.

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

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

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

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

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

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

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

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

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

3.7 Expressivity and Flexibility

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

3.7.1 Role-based Access Control Models

In RBAC [50], a role is a collection of permissions, and a permission is a pair (object, right) implying the right to the object. A role can be assigned to a user by an administrator or a security officer. A user can be assigned to a set of roles. In a session, a user activates a subset of his roles and obtains all the permissions associated with these activated roles. Roles may be organized in a partial order hierarchy, in which high-level roles (senior roles) inherit the permissions assigned to low-level roles (junior roles). RBAC can be expressed as pre-authorization models in UCON, in which user-role assignments can be regarded as subject attributes, permission-role assignments can be regarded as object attributes, and the partial order relation between roles in role hierarchy is expressed by attribute predicates.

Example 12 Consider an RBAC1 model [50] where all roles R are in a partial order hierarchy with respect to domination relation $\geq$. A subject (a user in RBAC1) has an attribute $actRole$ with value a subset of R, the activated roles in a session. An object has

an attribute $perRole$ with value a set of pairs $(role, r)$ where r is a right. A permission (o, r) is assigned to a $role$ iff $(role, r) \in o.perRole$. The predicate $rpa_r(role, o)$ is true if there exists $role'$ such that $role \geq role'$ and $(role', r) \in o.perRole$.

The usage control policy for RBAC1 is expressed by:

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

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

RBAC with constraints can also be expressed with a UCON model. There are many types of constraints that can be defined in RBAC, such as mutually exclusive roles, cardinality, prerequisite roles, etc. [50]. With appropriate attributes defined for subjects and objects, we can specify RBAC models with constraints using UCON.

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

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

   $preupdate(s.actRole) : s.actRole' = s.actRole \cup \{role_1\}$

This rule specifies that the permission (s, o, r) can be granted if $role_1$ is in the subject's $asgRole$ but not in $actRole$ (i.e., $role_1$ is assigned to s but not activated), $rpa_r(role_1, o)$ is

true, and $role_3$ is not in the value of the attribute $actRole$ of the subject. The $permitaccess$ action implies a pre-update action of the subject's $actRole$ attribute by adding $role_1$ to it. □

3.7.2 Chinese Wall Policy

The original Chinese Wall policy [15] prevents information flow between companies in conflict of interest. More generally, if a subject accesses an object in a conflict-of-interest set, then this subject cannot access any other object in this set in the future. We define an attribute to store the usage history of a subject: each time this subject generates an access request to an object, this attribute is checked and the authorization decision is determined by the history. In the meantime this attribute is updated to record this access information if the access request is approved. We show the policy with the following example.

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

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

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

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

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

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

The first one is a $preA_0$ policy, which specifies that when a subject wants to access an object accessed before, the access request is approved and there is no update. The second one is a $preA_1$ policy because of the update of the subject's attributes. Specifically, if an object's conflict set is not in a subject's ac list, this subject can access this object, and both ac and ao must be updated before the access. Note that in this system there are two policies for the permission $(s, o, read)$. In a real access period, only one of them is satisfied, as we mentioned in Section 3.3.1. □

3.7.3 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 [40] is presented to capture the attribute mutability property in DSoD. Specifically, an object attribute is defined to store the history of the subjects accessing this object. Here we present a simple example of object-based DSoD from [54].

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

The policies for $prepare$ and $issue$ are specified as follows, respectively.

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

   $preupdate(o.preparer) : o.preparer' = s.sid$

2. $permitaccess(s, o, issue) \rightarrow$

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

   $preupdate(o.issuer) : o.issuer' = s.sid$

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

3.7.4 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 3.3. With the high watermark property, the security clearance can be updated as a result of the user's access actions, and this update has to follow some predefined policies. We show this property in MAC as a $preA_1$ model.

Example 16 Suppose L is a lattice of security labels with relation $\geq$. A subject has two attributes, $clearance$ to represent the current label, and $maxClear$ to represent the maximum clearance label. An object has one attribute, $classification$. All these attributes have as value domain the lattice L. The authorization policy for $read$ is:

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

   $(s.maxClear \geq o.classification)\big) \wedge \blacklozenge preupdate(s.clearance)$

   $preupdate(s.clearance) : s.clearance' = LUB(s.clearance, o.classification)$

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

3.7.5 Hospital Information Systems

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

Example 17 Suppose that a doctor (s) can perform (r) a particular operation (o) only if he has operated more than 3 times before[2]. This can be expressed as a $preA_1$ model. The total times of the operations that a doctor has performed is stored as the subject attribute exp. The policy is:

1. $permitaccess(s, o, perform) \rightarrow$

   $\blacklozenge\big(tryaccess(s, o, perform) \wedge (s.role = doctor) \wedge (s.exp > 3)\big) \wedge \blacklozenge preupdate(s.exp)$

   $preupdate(s.exp) : s.exp' = s.exp + 1$ □

Example 18 In this example, a doctor can perform an operation on a patient only if the patient agrees to it on a consent form. This agreement is an obligation to be completed before the operation, where the patient is the obligation subject, and the consent is the

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

obligation object. This model can be expressed by a combination of $preA_0$ and $preB_0$. The policy is:

1. $permitaccess(s, o, operate) \rightarrow$

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

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

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

1. $permitaccess(s, o, operate) \rightarrow$

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

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

3.8 Related Work

Bertino et al. [8–10] introduce a temporal authorization model for database management systems. In this model, a subject has permissions 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. [20] 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. [53] apply interval temporal logic to express and compose access control polices, and Hansen and Sharp [22] introduce an approach for the analysis of security protocols using interval logic. The main difference in our approach is that we focus on the atomic actions and temporal properties during a single usage process, while their approaches focus on a higher level of system policies or security protocols.

Joshi et al. [27] 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 [28] 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. [12, 13] 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 [16] 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.

3.9 Summary

In this chapter I have developed a formal model of UCON with temporal logic of actions. A model is given by a set of system states in a single usage process, specified by a set of subjects and their attributes, a set of objects and their attributes, and the system attributes. The authorization predicates are built from subject and object attributes. Actions are the state transitions of the system, including usage control actions to update attributes and accessing status of a usage process, and obligation actions that have to be satisfied before or during an access. Conditions are predicates on system attributes. Temporal formulae represent usage control policies and are built from authorization predicates, actions, and system predicates. I prove that a fixed set of scheme rules can be used in general UCON policy specifications with soundness and completeness properties. The flexibility of the policy language is illustrated by expressing policies for various applications. The powerful specification capability of the extended TLA strengthens UCON with precise modeling and specification.

Chapter 4: Expressive Power

The policy specification flexibility of UCON has been conceptually shown in previous work and formally illustrated in the previous chapter. This chapter studies the expressive power of UCON by simulating some traditional access control models. For this purpose a formal UCON model is defined to formalize the accumulative effect of a usage process. The relative expressive power of UCON authorization models (UCON$_A$) and UCON obligation models (UCON$_B$) are studied.

In UCON condition models, a usage control decision is determined by some environmental restrictions dependant on system attributes. Since how system attributes change is not captured in UCON core models, the expressive power cannot be compared with other models.

4.1 Formal Model of UCON$_A$ and UCON$_B$

The logical model of UCON developed in Chapter 3 can precisely capture the new features of UCON, such as the attribute mutability and the decision continuity, but it is not appropriate to compare its expressive power to that of other access control models. The main reason is that the logical model specifies the detailed state change of the system in a single usage process, while for the expressive power, the overall effect of a usage process needs to be formulated. This is further motivated by the safety analysis of UCON, since the safety problem focuses on the permission propagation as the accumulative result of a sequence of

usage processes.

As another reason, the creating and destroying of subjects and objects are not formulated in the logical model developed in Chapter 3, since the logical model focuses on the temporal characteristics of the system state in a single usage process, and the accessing subject and the target object must exist in the system before the access request, as assumed in Section 3.3.1, that is, the logical model is developed to specify policies for UCON core models, where creating and destroying of subjects and objects are not included. Practically, all objects except the objects in the initial state are created in a system, and an object can be destroyed by a subject under particular circumstances. For example, in UCON, a derivative object is "derived from the original work" [38, 39]. "To provide mutual protection on the rights of all involved subjects (consumer, provider, and/or identifiee subjects), just like the original object, these derivative objects also have to be considered as target objects and must hold UCON properties and relations with other components" [38, 39]. That is, an object can be created from an existing object and includes some information of the original object, and needs to be protected. Usage log and payment information are typical derived objects in UCON. Although previously mentioned, the creating and destroying of subjects and objects are not included in UCON core models. To make the model more practical and complete to express policies for real systems, the policies of what kind of subjects and objects can be created, and how a subject or an object can be destroyed should be captured by a UCON model.

Because of these reasons, in this chapter a new formal model is proposed to capture the global effect of a usage process and the accumulative result of a sequence of usage

processes. Specifically, a single usage process is atomic, and all usage processes are serialized in a system. By serialized processes we mean that there is no interference between any two usage processes, so that the net effect is as though the individual usage processes executed serially one after another. We do not specify precisely how the serialization is achieved, since there are many standard techniques known for this purpose. The details of how to achieve serialization is an implementation-level issue as opposed to a model-level issue. Based on this, a set of policies are defined to specify the authorization predicates for usages, and sequences of primitive actions as the side-effect results. Also, policies for creating and destroying subjects and objects are defined.

This section presents the formal definition of UCON$_A$ (specifically, $preA$) and UCON$_B$ (specifically, $preB$). Some components of the models are introduced in Chapter 3, but for the consistency and completeness of the presentation they are re-defined in this chapter.

4.1.1 UCON$_A$

Subjects, Objects, and Rights

The subject, object and right abstractions are well known in access control. Generally speaking, a subject is an active object that can invoke some access requests or execute some permissions on another object, such as a process that opens a file for reading. A subject, in turn, can be accessed by another subject, e.g., a process can be created, stopped or killed by another process. Following the general concepts in traditional access control models, we consider the set of subjects in UCON$_A$ to be a subset of the set of objects. The objects that are not subjects are called pure objects. We require that each object is specified with an identity, called name, which is unique and cannot be changed, and cannot be reused

after the object is destroyed in the system. This unique name in many cases will not be the identity of a user. For example, a process executing on behalf of a user will have a process identity and not a user identity.

Rights are a set of privileges that a subject can hold and execute on an object, such as $read$, $write$, $pause$, etc. In access control systems, a right enables the access of a subject to an object in a particular mode, referred to as a permission. Formally, a permission is a triple (s, o, r), where s, o, r are a subject, object, and right, respectively. In UCON_A, a permission is enabled by an authorization rule in a policy.

The set of subjects, objects, and rights are denoted as S, O, and R, respectively, where $S \subseteq O$.

Attributes, Values, and States

Each object is specified with a non-empty and finite set of attributes. An attribute of an object is denoted as $o.a$ where o is the object name (i.e., the object's unique identity) and a is the attribute name. Note that an object name without any attribute specified denotes its identity attribute. Without loss of generality, we assume that in a system, every object has the same fixed set of attribute names ATT. The domain of the attribute a is denoted as $dom(a)$, and we assume that for $a \in ATT$, $null \notin dom(a)$.

An assignment of an attribute maps its attribute name to a value in its domain, denoted as $o.a = v$, where $v \in dom(a) \cup \{null\}$. The set of assignments for all objects' attributes collectively constitute a state of the system.

Definition 3. *A* system state*, or* state*, is a pair* (O, σ)*, where* O *is a set of objects, and* $\sigma : O \times ATT \rightarrow \bigcup_{a \in ATT} dom(a) \cup \{null\}$ *is a function that assigns a value or* $null$ *to*

each attribute of each object, where $\sigma(o, a) \in dom(a) \cup \{null\}$.

Predicates

Definition 4. *A predicate* $p(s, o)$ *is a boolean-valued polynomially computable function* [1] *built from a set of a subject* s*'s and an object* o*'s attributes and constants.*

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 this predicate. Similar to the previous chapter, unary and binary predicates can be defined based on a single object or two different objects.

Primitive Actions

A protection system evolves by the activities of the subjects, such as requesting and performing one or a sequence of accesses, which in turn may generate new objects in the system, or update the values of attributes corresponding to a set of usage control policies (defined shortly). Three kinds of primitive actions are defined in UCON_A.

Definition 5. *A* Primitive action *(or simply* action*) is a state transition of a system. Three primitive actions of UCON*$_A$ *are defined as in the Table 4.1, where* $t = (O, \sigma)$ *and* $t' = (O', \sigma')$ *are the states before and after a single primitive action.*

A $createObject$ action introduces a new object into the system, and requires that the new object not be in the system before the creation. Each attribute of the newly created object has the default value of $null$. Normally a $createObject$ is followed by

[1] As a predicate takes at most two parameters of objects, the function is polynomial in the number of object attributes and the size of their value domains.

Actions	Conditions	New States
$createObject\ o'$	$o' \notin O$	$O' = O \cup \{o'\}$ $\forall o \in O, a \in ATT, \sigma'(o.a) = \sigma(o.a)$ $\forall a \in ATT, \sigma'(o'.a) = null$
$destroyObject\ o$	$o \in O$	$O' = O - \{o\}$ $\forall o \in O', \forall a \in ATT, \delta'(o.a) = \delta(o.a)$
$updateAttribute\ o.a$: $o.a = v'$	$o \in O$, $a \in ATT$ $v' \in dom(a) \cup \{null\}$	$O' = O$ $\forall ent \in O, att \in ATT$, $\sigma'(ent.att) = \sigma(ent.att)$ if $ent \neq o$ and $att \neq a$, $\sigma'(o.a) = v'$

Table 4.1: Primitive actions

$updateAttribute$ actions to assign values to its attributes. The $destroyObject$ removes an existing object and its attributes from the system. For simplicity we assume that the identity of an object is unique during the system's life cycle, and cannot be reused even after the object is destroyed. The $updateAttribute$ action updates the value of an attribute $o.a$ from v to the new value v' which can be a constant, or the result generated by a polynomially computable function built from the old value v and other attribute values of the subject and object parameters of the policy.

UCON$_{\mathrm{A}}$ Policy

Satisfied predicates on attributes in UCON$_{\mathrm{A}}$ affect the system in two ways. First, a set of satisfied predicates can authorize a permission so that a subject can access an object with a particular right. Second, a set of satisfied predicates may authorize the system to move to a new state with a sequence of primitive actions, e.g., by creating a new object, or updating attribute values, which result from the allowed access. These actions, in turn, may make other predicates satisfied, and then enable other permissions and system state changes. The safety analysis of UCON$_{\mathrm{A}}$ (in the next chapter) focuses on the interactions between these two aspects, e.g, the permissions authorized by a system state and the state changes caused

by the actions.

Access authorizations and the state transitions are specified by a set of pre-defined policies in a system.

Definition 6. *A policy of UCON$_{\mathrm{A}}$ consists of a name, two parameter objects, an authorization rule, and a sequence of primitive actions as follows:*

policy_name*(s, o):*

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \rightarrow permit(s, o, r)$

act_1*;* act_2*;* $\ldots$*;* act_k

where s and o are the subject and object parameters; $p_1, p_2, \ldots, p_i$ are predicates based on s's and o's attributes and constants; $permit(s, o, r)$ is a predicate which indicates that a permission (s, o, r) is authorized by the system if $true$; act_1, act_2, $\ldots$, act_k are primitive actions that are performed on s or o or their attributes.

We assume that s is the active object in a policy, so it is the subject that attempts an operation requiring the right r on the target object o. It is also possible to have $s = o$, wherein a subject performs some operation on itself.

A policy includes two parts. The first part is an authorization rule consisting of a conjunction of attribute predicates, called the *condition* of the policy, followed by a $permit$ predicate implied by the condition. The second part is a sequence of primitive actions, called the *body* of the policy. The first part specifies a permission authorized by the state of the system, while the second part is the side-effect of executing this permission, thereby changing the current state of the system to a new state. Note that there may be policies that have no actions but only authorization rules, e.g., similar to the $preA_0$ in the previous

chapter. Enforcing a policy without actions causes no state transition of the system. In any state, a permission that is not *permitted explicitly* by a policy is denied by default. In general the UCON$_A$ model only considers positive permissions.

As it shows, a policy here includes the authorization predicates for a usage and the resulting actions. Instead of using the $premitaccess$ action as that in the logical model, the $permit$ predicate indicates whether an access is permitted or not, and hides the individual actions in the usage process, that is, the new policy specifies the overall effects on the system state for a usage process. This approach captures the essential aspect of system state transitions and permission propagations caused by the attribute mutability of UCON, while maintaining the simplicity of the policy specifications.

Note that by the policy definition we assume that all the authorization predicates in a policy are considered as pre-authorizations, and all the updates as post-updates. That is, the UCON$_A$ model defined in this section is $preA_3$. As all usage processes are serialized in a UCON$_A$ system, and a policy captures the overall effects of the system state after a usage process, the updates in a policy can also be considered as pre-updates or ongoing updates, which would make the model $preA_1$ or $preA_2$, respectively. All expressive power and safety analysis results derived for $preA_3$ thereby also hold for $preA_1$ and $preA_2$. For the sake of simplicity, we assume without loss of generality that the UCON$_A$ model considered in this chapter and next chapter is a $preA_3$ model.

Definition 7. *A policy is a* creating policy *if it contains a* $createObject$ *action in its body; otherwise, it is* non-creating.

A policy is enforced when an access requested is generated. Therefore, at least one of its parameters exists in the system before the request, and a creating policy can contain at

most one $createObject$ action. Without loss of generality, we assume that, in a creating policy, the first parameter s which is the *parent* object must exist before the actions, and o is created as a *child* object. Hence, UCON$_A$ is a single-parent creation model. Also, we can assume that, in a policy, there is at most one update action for any attribute of an object, since multiple updates on the same attribute can be reduced to a single update with the value of the last one. Negated predicates are not explicitly required since we can always define new predicates equivalent to negated predicates. For example, instead of $\neg(s.credit > \$1000)$, we use $(s.credit \leq \$1000)$. Similarly, disjunction of predicates is not explicitly required since it can be expressed by a set of individual policies, one for each component of the disjunction.

A policy is enforced by replacing the two parameters with a pair of actual subject and object names when the subject generates an access request on the object with a particular right. If the condition of the policy and all the conditions for the primitive actions are satisfied, then the permission is authorized, and all the primitive actions are performed. Otherwise, the permission is not granted, and the system does not change state. As mentioned, we assume that all accesses in a system are serialized, and that the enforcement of each policy is atomic, either an access is granted and all the primitive actions are completed, or the system state does not change.

Example 20 Suppose that a document can only be issued by a *scientist* (with role sci). For *anonymous* users, this document can only be read 10 times. We define the available times ($readTimes$) as an object attribute. Each time an anonymous user is authorized to read a document, this attribute is updated by decreasing it by one. The policies in this application are:

$create\_doc(s, doc)$:

$(s.role = sci) \rightarrow permit(s, doc, create)$

$createObject\ doc$

$updateAttribute$: $doc.readTimes' = 10$

$read\_doc(s, doc)$:

$(s.role = anonymous) \wedge (doc.readTimes > 0) \rightarrow permit(s, doc, read)$

$updateAttribute$: $doc.readTimes' = doc.readTimes - 1$

The first creating policy specifies that a subject in the role sci can *create* a new document, and the $readTimes$ attribute of this new object is set to 10. In the second policy, a subject with role $anonymous$ can be authorized to $read$ a document if its $readTimes$ attribute is positive; as a result of this permission, the value of the attribute $readTimes$ is decreased by one. □

UCON$_{\mathrm{A}}$ Protection System

A formal representation of a UCON$_{\mathrm{A}}$ system can be defined with the basic components that we have introduced.

Definition 8. *A UCON$_{\mathrm{A}}$* scheme *is a 4-tuple (ATT, R, P, C), where ATT is a finite set of attribute names, R is a finite set of rights, P is a finite set of predicates, and C is a finite set of policies. A UCON$_{\mathrm{A}}$* protection system *(or simply* system*) is specified by a UCON$_{\mathrm{A}}$ scheme and an initial state (O_0, σ_0).*

Definition 9. *Given a UCON$_A$ system, the* permission function *of a state $t = (O, \sigma)$ is $\rho_t : O \times O \rightarrow 2^R$, and if $r \in \rho_t(s, o)$, then in the state t, the subject s can access the object*

o with the right r according to at least one policy.

The function ρ_t maps a pair (subject, object) [2] to a set of generic rights, according to their attribute-value assignments in state t and the set of policies in the scheme. In a particular state, the value of $\rho_t(s, o)$ can be determined by trying each policy in the scheme with the attribute-value assignments of s and o. With the finite number of predicates in a policy and the finite number of policies in a scheme, the complexity of computing ρ_t for a pair (s, o) is $\mathcal{O}(|P| \times |C|)$.

Definition 10. *For two states (O_t, σ_t) and $(O_{t'}, \sigma_{t'})$ of a system:*

- *$t \twoheadrightarrow_c t'$ ($c \in C$) if there exist a pair of objects (o_1, o_2) ($o_1 \in O_t$) such that the policy $c(o_1, o_2)$ can be enforced in the state t and the system state changes to t';*
- *$t \twoheadrightarrow_C t'$ if there exist a $c \in C$ such that $t \twoheadrightarrow_c t'$;*
- *$t \rightsquigarrow_C t'$ if there exist a sequence of states $t_1, t_2, \ldots, t_n$ such that $t \twoheadrightarrow_C t_1 \twoheadrightarrow_C t_2 \cdots \twoheadrightarrow_C t_n \twoheadrightarrow_C t'$.*

A transition history *from state t to state t' is denoted as $t \rightsquigarrow_C t'$, or simply $t \rightsquigarrow t'$.*

4.1.2 UCON$_B$

A usage control decision in UCON$_B$ is determined by obligation actions. The subject and object attribute predicates, system states, primitive actions, permission function, and system state transition history are defined as in UCON$_A$. Besides that, a UCON$_B$ policy includes one or more obligation actions.

[2] Note that a subject is also an object in the model.

Definition 11. *An* obligation action *is represented by a boolean-valued expression built from an obligation name, an obligation subject, and an obligation object.*

Definition 12. *A policy of UCON*$_{\mathrm{B}}$ *consists of a name, a set of parameter objects, an obligation rule, and a sequence of actions as follows:*

policy_name*(*$s, o, sb_1, ob_1, sb_2, ob_2, \ldots, sb_j, ob_j$*):*

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \wedge b_1(sb_1, ob_1) \wedge b_2(sb_2, ob_2) \cdots \wedge b_j(sb_j, ob_j) \rightarrow permit(s, o, r)$

act_1*;* act_2*; . . . ;* act_k

where s *and* o *are the requesting subject and the target object of the access;* $sb_1, ob_1, \ldots, sb_j, ob_j$ *are the obligation subjects and objects for the obligation* $b_1, \ldots, b_j$*, respectively;* $p_1, \ldots, p_i$ *are predicates based on the attributes of all parameter subjects and objects* [3]*;* $permit(s, o, r)$ *is the decision predicate;* $act_1, act_2, \ldots, act_k$ *are primitive actions that are performed on* s *or* o *or their attributes.*

A UCON$_B$ policy includes two parts: an obligation rule consisting of a conjunction of predicates and actions, which is called the *condition* of the policy, followed by a $permit$ action implied by the condition, and a sequence of primitive actions, which is called the *body* of the policy. Attribute predicates in a UCON$_B$ policy identify what kind of obligations are required for the usage. For example, a child's downloading of a movie requires his/her parent to sign an agreement. A predicate is needed to bind the requesting subject

[3]Because of the informal conceptual model of UCON preB given in [39], it is ambiguous how to fully formulate it. We describe one approach here. Specifically, the predicates in a UCON$_B$ policy capture the predicate $getPreOBL$ in Definition 8 in [39], but is more powerful than needed just for this purpose. For instance, they allow the testing of attributes of obligation subjects and obligation objects, which is not used in the subsequent construction of this chapter.

and obligation subject. The predicates in a UCON$_B$ policy are not for authorization reason, e.g., the relation of the obligation subject and the requesting subject cannot enable the permission of the access.

Considering the general case that an obligation subject can be the requesting subject, and an obligation object can be the target object, we write a UCON$_B$ policy as follows.

policy_name($o_1, o_2, \ldots, o_m$):

$p_1(o_{sp1}, o_{op1}) \wedge p_2(o_{sp2}, o_{op2}) \wedge \cdots \wedge p_i(o_{spi}, o_{opi}) \wedge b_1(o_{sb1}, o_{ob1}) \wedge b_2(o_{sb2}, o_{ob2}) \cdots \wedge$

$b_j(o_{sbj}, o_{obj}) \rightarrow permit(o_1, o_2, r)$

$act_1; act_2; \ldots; act_k$

where $sp1, op1, \ldots, spi, opi$ and $sb1, ob1, \ldots, sbj, obj$ are integers from 1 to m. By convention, o_1 is the accessing subject, and o_2 is the target object. In a creating policy, o_2 is the child object. The actions $act_1, \ldots, act_k$ are limited to o_1 and o_2.

The obligations defined here are all pre-obligations, and all updates are post-updates, that is, the model defined in this subsection is $preB_3$. Similar to UCON$_A$, as all usage processes are serialized in a UCON$_B$ system, and a policy captures the overall effects of the system state after a usage process, the updates in a policy can also be considered as pre-updates or ongoing updates, which would make the model $preB_1$ or $preB_2$, respectively. All expressive power and safety analysis results derived for $preB_3$ thereby also hold for $preB_1$ and $preB_2$. For the sake of simplicity, we assume without loss of generality that the UCON$_B$ model considered in this chapter and next chapter is a $preB_3$ model.

Example 21 Alice (s) can *download* an online movie (o) only if her parent (sb) signs an *agreement* (ob). Alice's $credit$ is reduced by subtracting the movies's $value$. The policy is:

$download\_movie(s, o, sb, agreement)$

$(s.parent = sb) \wedge sign(sb, agreement) \rightarrow (s, o, download)$

$updateAttribute$: $s.credit' = s.credit - o.value$ □

Example 22 Bob (s) can *open* an email (o) only after he clicks the *acknowledgement* button (ob). The email's *flag* attribute is changed to "read". The policy is:

$open\_email(s, o, ack)$

$click(s, ack) \rightarrow (s, o, open)$

$updateAttribute$: $o.flag' =$ "$read$" □

Definition 13. *A UCON*$_{\mathrm{B}}$ scheme *is a 5-tuple* (ATT, R, P, B, C)*, where* ATT *is a set of attribute names,* R *is a set of rights,* P *is a set of predicates,* B *is a set of obligation actions, and* C *is a set of policies. A UCON*$_{\mathrm{B}}$ protection system *(or simply* system*) is specified by a UCON*$_{\mathrm{B}}$ *scheme and an initial state* (O_0, σ_0).

4.2 Expressive Power of UCON$_A$

In this section, I first informally show the expressive power of UCON$_A$ with a DRM application. Then I formally demonstrate the expressive power of UCON$_{\mathrm{A}}$ by simulating Single-Object Typed Access Matrix model (SO-TAM), which is known to have equivalent expressive power to TAM [21]. This proves that UCON$_A$ at least has the expressive power of TAM. The construction is easily extended to show the same result between UCON$_A$ and augmented TAM (ATAM) as discussed at the end of this section.

4.2.1 A UCON$_A$ Model for iTunes-like Systems

Apple iTunes is a popular online music downloading and legally sharing service. A user can register and order music files from an iTunes server, and authorize a set of platforms

to play these files. In this section I specify the core functions of iTunes-like systems with UCON$_A$ polices. This is a simplified version of the actual iTunes system.

Subjects, Objects, and Attributes

The objects in an iTunes-like system include users, iTunes servers, platforms, and music files. The corresponding attributes are defined in Table 4.2.

Objects	Attribute	Value
user	$registered$	a boolean value to indicate if a user is registered or not
	$credit$	a numerical value of the credit balance of a user's account
	$orderList$	a set of music files that a user has ordered
	$platformList$	a set of platforms that a user authorizes to play music
iTunes server	$regUsers$	a set of registered users (e.g., accounts)
platform	$authorizedBy$	a user that authorizes this platform
	$localList$	a set of music files that stored locally in a platform
music file	$owner$	the user that owns this file
	$price$	a numerical value of the music file's price

Table 4.2: Attributes in UCON$_A$ for iTunes-like Systems

Policies

Before ordering and downloading a music file, a user needs to register with an iTunes server. The following policy captures this.

$user\_register(s, u)$:

$true \rightarrow permit(s, u, register)$

$createObject\ u$;

$updateAttribute:\ s.regUsers' = s.regUsers \cup \{u\}$;

$updateAttribute:\ u.registered' = true$;

$updateAttribute:\ u.platformList' = \emptyset$;

$updateAttribute: \ u.orderList' = \emptyset$;

$updateAttribute: \ u.credit' = 0.00$;

where s is an iTunes server and u is a user. After registration, a new object (the user) is created, and the attributes are set to their initial values.

An order of a music file can be specified as the following policy.

$order(u, m)$:

$(u.registered = true) \wedge (u.credit \geq m.price) \wedge (m \notin u.orderList) \rightarrow$

$permit(u, m, order)$

$updateAttribute: \ u.orderList' = u.orderList \cup \{m\}$;

$updateAttribute: \ m.owner' = u$;

$updateAttribute: \ u.credit' = u.credit - m.price$;

where u is a user and m is a music file. This policy checks if the user has sufficient credit and if the music file has not been previously purchased by the user, in which case the user can place the order. As a result, the user's $orderList$ attribute and the music file's $owner$ attribute are updated, and the user's credit is decreased by the price of the music file (e.g., $\$0.99$ in current iTunes service).

To play downloaded music, a user needs to authorize a platform. By default, a user can authorize at most five platforms in current iTunes service. Conversely a user can also de-authorize a platform. The following policies specify these.

$authorize\_platform(u, p)$:

$(u.registered = true) \wedge (|u.platformList| < 5) \wedge (p \notin u.platformList) \rightarrow$

$permit(u, p, authorize)$

$updateAttribute: \ u.platformList' = u.platformList \cup \{p\};$

$updateAttribute: \ p.authorizedBy' = u;$

$deauthorize\_platform(u, p)$:

$(u.registered = true) \wedge (p \in u.platformList) \rightarrow permit(u, p, deauthorize)$

$updateAttribute: \ u.platformList' = u.platformList - \{p\};$

$updateAttribute: \ p.authorizedBy' = null;$

where u is a user and p is a platform.

Finally, a platform can play a music file if it is authorized by a user, and the music file is owned by the same user, as the following policy indicates.

$play(p, m)$:

$(p.authorizedby \neq null) \wedge (m.owner \neq null) \wedge (p.authorizedby = m.owner) \rightarrow$

$permit(p, m, play)$

where p is a platform and m is a music file.

Note that these UCON$_A$ policies only capture the core functions of an iTunes service. Some actions are regarded as external or administrative actions and not included in this UCON$_A$ model. Among them,

- a user's credit is increased by charging the user's credit card or redeeming a gift certificate;
- a music file is exported to external devices, e.g., by burning a CD. Since the music file is changed to a different format (e.g., mp3 or wma) after burning, the licence

information is lost. Thus this new file is considered as a different object, and its usage is not captured in this model.

4.2.2 TAM and SO-TAM

The formal study of the expressive power of UCON is performed by simulating traditional access control models using UCON. Since the access matrix model is well studied and widely applied, in this work I compare the expressive power of UCON$_A$ and some access matrix models. Specifically, I simulate SO-TAM, which has the same expressive power as TAM, with UCON$_A$.

The key innovation of TAM [46], the typed access matrix model, is to introduce strong types of subjects and objects into the traditional access matrix model formalized by Harrison, Russo, and Ullman (HRU) [23], and the strong type concept is motivated by the schematic protection model (SPM) [45]. In TAM, rights are distributed through a matrix, in which each subject is represented by a row and a column, while a pure object is represented by a column. An access control policy is enforced by checking the presence of corresponding right in the particular cell; e.g., the cell $[s, o]$ contains that rights that s possesses for o. Each object is created to be of a particular type that cannot be changed. A TAM scheme includes a fixed set of rights RGT, a fixed set of types TYP, and a fixed set of commands COM. A TAM system state is defined by a tuple (SUB, OBJ, f, M), where SUB is a set of subjects, OBJ is a set of objects, $SUB \subseteq OBJ$, f is a type function $f : OBJ \rightarrow TYP$, and $M(SUB, OBJ)$ is a configuration, where M is a matrix, and $M[s, o] \subseteq RGT$ is the content of the cell $[s, o]$. A TAM system is specified by a TAM scheme and an initial state (SUB_0, OBJ_0, f_0, M_0).

The presence of some rights in a matrix cell not only gives the subject those rights on the object, but may authorize a subject to change the matrix by entering or removing rights in some cells, or creating and deleting rows and columns. This change in a system state is conducted by executing commands in TAM. A command in TAM has the following general format.

Command $\alpha(X_1 : t_1, X_2 : t_2, \ldots, X_k : t_k)$

if $r_1 \in [X_{s1}, X_{o1}] \wedge r_2 \in [X_{s2}, X_{o2}] \wedge \ldots r_m \in [X_{sm}, X_{om}]$

then

op_1; op_2; $\ldots$; op_n

end

where $X_1, X_2, \ldots, X_k$ are subject or object parameters whose type are t_1, t_2, $\ldots$, t_k, respectively; r_1, r_2, $\ldots$, r_m are rights; $s1$, $s2$, $\ldots$, sm and $o1$, $o2$, $\ldots$, om are integers between 1 and k; op_1, $\ldots$, op_n are primitive operations. The if part of the command is called the condition of α. A command is invoked by substituting actual subjects and objects of the appropriate types as parameters. The operations are executed sequentially if the condition is true. There must be at least one subject type in a TAM command. One or more subjects collectively execute the command. Detailed specification of who actually invokes the command is not explicitly given in TAM. There are six primitive operations in TAM: enter r into $[s, o]$; create subject s of type t_s; create object o of type t_o; delete r from $[s, o]$; destroy subject s; destroy object o; The first three are regarded as monotonic operations, while the last three are non-monotonic operations.

Single-Object TAM (SO-TAM) is restricted from TAM such that all primitive operations in a command are performed on a single object. That means, SO-TAM can check the

presence of rights in any number of cells, but can only modify cells along a single column in the matrix. Furthermore, a single SO-TAM command can create or destroy at most one object, which is also the object whose cells are modified. The creation operation (if any) must therefore occur as the first in the sequence of operations and the destroying operation (if any) as the last. Without loss of generality, we can assume that, for a command $\alpha(X_1 : t_1, X_2 : t_2, \ldots, X_k : t_k)$, X_k is the object on which all the operations are performed. If the command α contains a "create object" operation, then X_k is the object created.

Example 23 Of the two commands below, $reviewdoc$ is a SO-TAM command, while $share\_ownership$ is not, since the two primitive operations in this command are performed on two different objects.

command $reviewdoc(s : sci, so : sec\_off, po : pat\_off, o : doc)$

if $own \in [s, o]$ **then**

enter $review$ in $[so, o]$;

enter $review$ in $[po, o]$;

end

command $share\_ownership(s1 : t_{s1}, s2 : t_{s2}, o1 : t_{o1}, o2 : t_{o2})$

if $own \in [s1, o1] \wedge own \in [s2, o2]$ **then**

enter own in $[s1, o2]$;

enter own in $[s2, o1]$;

end □

4.2.3 Simulating SO-TAM with UCON$_A$

To compare the expressive power of UCON$_A$ and SO-TAM, we simulate a general SO-TAM system with a UCON$_A$ system. A construction similar to the one in [21] is used in this simulation.

Theorem 3. *UCON$_A$ is at least as expressive as SO-TAM.*

Proof. For a SO-TAM system $\mathcal{T}$ with scheme (RGT, TYP, COM) and an initial state (SUB_0, OBJ_0, f_0, M_0), we construct a UCON$_\mathrm{A}$ system $\mathcal{U}_a$ with scheme (ATT, R, P, C) and an initial state (O_o, AM_0) to simulate it. Specifically, $R = RGT \cup \{create, destroy, null\}$, where $null$ is a special right that is used for the intermediate policies during the simulation. We can simply consider it as an unknown right without any practical meaning; We can also ensure that $create \notin RGT$ and $destroy \notin RGT$. P and C are attribute predicates and policies which are defined during the simulation shown shortly. ATT is a set of attributes defined below.

1. $type \in TYP \cup \{syn\}$: the type of a subject or object where $syn \notin TYP$.

2. $acl \subseteq SUB \times RGT$: the access control list of an object.

3. $cc \in \{token\} \cup \{\alpha_i | \alpha \in COM, 0 \leq i \leq K, 1 \leq j \leq l\}$, where K is the maximum number of parameters in all the commands, and l is the number of commands in $\mathcal{T}$. The value of this attribute is the current command that the object is involved in. An object whose cc has value α_i is involved in α as the ith parameter. For $i = 0$, α_0 is a value used only for a special object SYN. The value $token$ specifies that no ongoing command is being performed with this object, so the object is available for a new command to start.

4. $param \subseteq \{po_i\}$, $1 \leq i \leq K$. This attribute records the parameter objects in a simulation. $po_i \in SYN.param$ indicates that the ith parameter object is present.

5. $present \subseteq \{prst_i\}$, $1 \leq i \leq M$, where M is the maximum number of condition components in all the commands. This attribute records the condition information in a command. $prst_i \in X_{oi}.present$ iff $r_i \in [X_{si}, X_{oi}]$ and this is the ith condition part in a command.

6. $opt \subseteq \{opt_i\}$, $1 \leq i \leq N$, attribute recording the primitive operations placed in a command, where N is the maximum number of primitive operations in all the commands.

7. $condition \in \{true, false\}$: attribute to specify whether all the condition parts are satisfied in a command.

8. $opt\_done \in \{true, false\}$: attribute to specify whether all the primitive operations have been performed in a command.

How the initial state (SUB_0, OBJ_0, f_0, M_0) of $\mathcal{T}$ is mapped into the initial state (O_o, AM_0) of $\mathcal{U}_a$ is shown next.

- $O_0 = OBJ_0 \cup \{SYN\}$
- $\forall o \in OBJ_0, o.acl = \{(s, r) | s \in SUB_0, r \in M_0(s, o)\}$
- $SYN.acl = \emptyset$
- $\forall o \in OBJ_0, o.type = f_0(o)$
- $SYN.type = syn$

- $\forall o \in O_0, o.cc = token$
- $\forall o \in O_0, o.param = \emptyset$
- $\forall o \in O_0, o.present = \emptyset$
- $\forall o \in O_0, o.opt = \emptyset$
- $\forall o \in O_0, o.condition = false$
- $\forall o \in O_0, o.opt\_done = false$

A right present in a cell enables the access right of the subject to the object, this is expressed as a set policies in $\mathcal{U}_a$, with the object's acl attribute without any actions, as shown below.

$policy\_phase0\_r$ (X_1, X_2):
$\big((X_1, r) \in X_2.acl\big) \rightarrow permit(X_1, X_2, r)$

Since we need a policy for each generic right, the number of policies in this set is $|RGT|$.

The commands in the TAM scheme are simulated one by one, where each of them is reduced to a linear number of policies on the number of parameters, condition components, and primitive actions in a command. As a policy involves only a subject and an object, the basic idea of this simulation is to create a special object SYN with type syn, and put the condition information of the command in the attributes of this special object by a set of policies. If all the conditions are satisfied, then the operations on the single object can be performed by another set of policies. With the existence of non-monotonic operations in SO-TAM commands, a condition in a command may not be satisfied if other commands

are executed to change the matrix at the same time. So we need a synchronization between commands such that all the commands are simulated serially. That is, we require a serialized simulation in which only one command is performed at one time. This can be done by checking and updating the attributes of SYN.

Let α be a command in $\mathcal{T}$ as the following:

Command $\alpha(X_1 : t_1, X_2 : t_2, \ldots, X_k : t_k)$

if $r_1 \in [X_{s1}, X_{o1}] \wedge r_2 \in [X_{s2}, X_{o2}] \wedge \ldots r_m \in [X_{sm}, X_{om}]$

then

$[create\ object\ X_k\ of\ type\ t_k]$;

...

$enter\ r_i\ into\ [X_{pi}, X_k]$;

...

$remove\ r_j\ from\ [X_{pj}, X_k]$;

...

$enter/remove\ r_n\ into/from\ [X_{pn}, X_k]$;

$[destroy\ object\ X_k]$;

end

where k is the number of parameters; m is the number of condition parts; n is the number of primitive operations except creating and destroying; $1 \leq i, j \leq n$; pi and pj are integers from 1 to k. The creating and destroying operations are optional in α. If there is any, the creating operation must be the first one, and the destroying operation must be the last one.

For this general command, five phases are used to simulate it in $\mathcal{U}_a$, each of which includes a bounded set of policies. We explain the details phase by phase.

Phase I: This phase initializes the simulation by the following policy with SYN.

α_phaseI_SYN (SYN, SYN):

$(SYN.cc = token) \rightarrow permit(SYN, SYN, null)$

$updateAttribute : SYN.cc' = \alpha_0$

This policy checks SYN's cc attribute. If the value is $token$, the execution of a new command can be started, and this attribute's value is updated to the command name. Note that there is only one object (SYN) in the system with syn type, so we do not need to check SYN's type.

After this policy, the simulation serially enforces the following policies for each parameter X_i in α, where $1 \leq i \leq k$.

$\alpha\_phaseI\_X_i\_t_i$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (po_i \notin SYN.param) \wedge (X.type = t_i) \wedge (X.cc = token) \rightarrow$

$permit(SYN, X, null)$

$updateAttribute : X.cc' = \alpha_i$

$updateAttribute : SYN.param' = SYN.param \cup \{po_i\}$

This policy updates the cc attribute of X to α_i to prevent interference from other commands. At the same time, the value α_i indicates that X is the ith parameter in α with type t_i. As each command requires exactly one parameter object of X_i, the presence of po_i in $SYN.param$ indicates this.

Note that if X_k is an object that is created in α, the policy between SYN and X_k is not included in this phase. So for a command with k parameters, there are $k + 1$ policies in this phase if there is no creating operation in this command; otherwise, there are k policies.

Since all the operations are performed on a single object, there is at most one creating action in a SO-TAM command.

Phase II: This phase includes a set of policies to check the condition parts in α. Specifically, for each condition part $r_j \in [X_{sj}, X_{oj}]$ where $1 \leq j \leq m$, we create two policies: one policy to update X_{oj}'s $present$ attribute, and another to update SYN's attribute based on this change.

$\alpha\_phaseII\_r_j\_X_{sj}\_X_{oj}$ (X_1, X_2):

$(X_1.cc = \alpha_{sj}) \wedge (X_2.cc = \alpha_{oj}) \wedge \big((X_1, r_j) \in X_2.acl\big) \rightarrow permit(X_1, X_2, null)$

$updateAttribute : X_2.present' = X_2.present \cup \{prst_j\}$

$\alpha\_phaseII\_r_j\_SYN\_X_{oj}$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (X.cc = \alpha_{oj}) \wedge (prst_j \in X.present) \rightarrow permit(SYN, X, null)$

$updateAttribute : SYN.present' = SYN.present \cup \{prst_j\}$

For a command with m condition parts, there are $2m$ policies in this phase.

Phase III: After the simulation in Phase II, all the condition parts have been recorded in SYN's $present$ attribute. In this phase there is one policy to create the object X_k (if α is a creating command) and update its attributes.

If α is a creating command, the creation must be the first operation in the command. The following simulates it.

$\alpha\_phaseIII\_SYN\_X_k\_create$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (prst_1 \in SYN.present) \wedge (prst_2 \in SYN.present) \wedge \cdots \wedge$

$(prst_m \in SYN.present) \rightarrow permit(SYN, X, create)$

$createObject\ X$

$updateAttribute : X.type' = t_k$

$updateAttribute : X.cc' = \alpha_k$

$updateAttribute : X.condition' = true$

$updateAttribute : SYN.condition' = true$

$updateAttribute : SYN.param' = SYN.param \cup \{po_k\}$

This policy first creates a new object with type t_k as the parameter X_k in α. At the same time, the $condition$ attribute of this new object is changed to $true$ since all the conditions parts have been satisfied.

If α is not a creating command, the following policy just updates X_k's $condition$ value based on the evaluations in Phase II.

$\alpha\_phaseIII\_SYN\_X_k$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (X.cc = \alpha_k) \wedge (prst_1 \in SYN.present) \wedge (prst_2 \in SYN.present) \wedge \cdots \wedge (prst_m \in SYN.present) \rightarrow permit(SYN, X, null)$

$updateAttribute : SYN.condition' = true$

$updateAttribute : X.condition' = true$

Phase IV: After Phase III, X_k's attribute has been updated according to the condition in α. In this phase, the primitive operations in α are simulated one by one. Basically, each operation except creating (included in Phase III if α has this) and destroying (included in phase V if α has this) is simulated by a policy in this phase. Specifically, for a primitive operation: $enter\ r_i\ into\ [X_{pi}, X_k]$, we have the following policy:

$\alpha\_phaseIV\_X_{pi}\_X_k\_enter\_r_i$ (X_1, X_2):

$(X_1.cc = \alpha_{pi}) \wedge (X_2.cc = \alpha_k) \wedge (X_2.condition = true) \rightarrow permit(X_1, X_2, null)$

$updateAttribute : X_2.acl' = X_2.acl \cup \{(X_1, r_i)\}$

$updateAttribute : X_2.opt' = X_2.opt \cup \{opt_i\}$

The first action in this policy updates X_2's acl attribute by adding (X_1, r) if X_1 and X_2 are the pith and kth parameters in α, respectively. The second action records the fact that this primitive operation has been completed.

For a primitive operation: $remove\ r_j\ from\ [X_{pj}, X_k]$, we have the following policy:

$\alpha\_phaseIV\_X_{pj}\_X_k\_remove\_r_j$ (X_1, X_2):

$(X_1.cc = \alpha_{pj}) \wedge (X_2.cc = \alpha_k) \wedge (X_2.condition = true) \rightarrow permit(X_1, X_2, null)$

$updateAttribute : X_2.acl' = X_2.acl - \{(X_1, r_j)\}$

$updateAttribute : X_2.opt' = X_2.opt \cup \{opt_j\}$

Similarly, the first action in this policy updates X_2's acl attribute by removing (X_1, r) if X_1 and X_2 are the pjth and kth parameters in α, respectively. The second action records the fact that this primitive operation has been completed.

For a command with n primitive operations except creating and destroying, there are n policies to simulate them in this phase.

Phase V: After all the operations (on the object X_k) have been completed, we need to reset the attributes of all the subjects and objects involved in this command for the next command. At first we need a policy to notice that all the operations, except destroying X_k if α includes it, have been completed.

$\alpha\_phaseV\_SYN\_X_k$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (X.cc = \alpha_k) \wedge (opt_1 \in X.opt) \wedge \cdots \wedge (opt_n \in X.opt) \rightarrow$

$permit(SYN, X, null)$

$updateAttribute : X.opt\_done' = true$

$updateAttribute : SYN.opt\_done' = true$

If α has a destroy operation, the destroy must be the last operation in the command. The following policy simulates it.

$\alpha\_phaseV\_SYN\_X_k\_destroy$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (X.cc = \alpha_k) \wedge (SYN.opt\_done = true) \rightarrow permit(SYN, X, destroy)$

$destroyObject\ X$;

$updateAttribute : SYN.param' = SYN.param - \{po_k\}$

After this step, all the primitive operations have been completed. We need to reset all the subjects' and objects' attributes for the next command. Similar to the policies in Phase I, a set of policies between SYN and the parameter entities are applied to reset their attributes as the following shows, , where $1 \leq i \leq k$

$\alpha\_phaseV\_SYN\_X_i\_Success$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (X.cc = \alpha_i) \wedge (SYN.opt\_done = true) \rightarrow permit(SYN, X, null)$

$updateAttribute : X.cc' = token$

$updateAttribute : X.present' = \emptyset$

$updateAttribute : X.opt' = \emptyset$

$updateAttribute : X.condition' = false$

$updateAttribute : X.opt\_done' = false$

$updateAttribute : SYN.param' = SYN.param - \{po_i\}$

This policy checks if all the primitive operations in α have been performed. From Phase IV, we can see that a primitive operation can be performed only if $SYN.condition$ is true,

that means, all condition parts in α are true. Note that if X_k is destroyed in α, then the corresponding resetting policy is not needed here. After all the attributes of the parameter objects have been reset ($SYN.param$ is empty), we need to reset the SYN's attributes.

$\alpha\_phaseV\_SYN\_Success$ (SYN, SYN):

$(SYN.cc = \alpha_0) \wedge (SYN.opt\_done = true) \wedge (|SYN.param| = 0) \rightarrow$

$permit(SYN, SYN, null)$

$updateAttribute : SYN.cc' = token$

$updateAttribute : SYN.present' = \emptyset$

$updateAttribute : SYN.opt' = \emptyset$

$updateAttribute : SYN.condition' = false$

$updateAttribute : SYN.opt\_done' = false$

After this policy, the system is ready to execute another command.

A resetting process may also be needed if any condition part is not satisfied with the actual parameters; i.e., in Phase II, if a condition part (say, $r_j \in [X_{sj}, X_{oj}]$) is not satisfied, then $prst_j \notin SYN.present$ and the simulation cannot go further. In this case, either a new parameter object can be captured, or the all the parameters and SYN can be reset for a rollback. The following policy can reset an object so that another object can be captured as the parameter, where $1 \leq i \leq k$.

$\alpha\_phaseV\_SYN\_X_i\_Rollback$ (SYN, X):

$(SYN.cc = \alpha_0) \wedge (SYN.condition = false) \wedge (X.cc = \alpha_i) \rightarrow permit(SYN, X, null)$

$updateAttribute : X.cc' = token$

$updateAttribute : X.present' = \emptyset$

$updateAttribute : SYN.param' = SYN.param - \{po_i\}$

This policy states that if $SYN.condition$ is false, the attributes of X_i can be reset, and po_i is removed from $SYN.param$ so that another object can be captured as parameter X_i. The value of $SYN.condition$ is false if either some policies in Phase II have not been executed, or at least one of the condition part checks in Phase II has failed so that the condition evaluation policy in Phase III is not executed. For the first case, the simulation of α can be restarted after resettings. For the second case, the condition of α is not satisfied with the object parameters. So they can be reset and other objects can be involved with the policies in Phase I. Since this policy requires $SYN.condition = false$, then $X.condition = false$, $X.opt = \emptyset$, and $X.opt\_done = false$, and there are no updates for them.

SYN's attributes can also be reset with the following policy for rollback when $SYN.param$ is empty.

$\alpha\_phaseV\_SYN\_Rollback$ (SYN, SYN):

$(SYN.cc = \alpha_0) \wedge (SYN.condition = false) \wedge (|SYN.param| = 0) \rightarrow$

$permit(SYN, SYN, null)$

$updateAttribute : SYN.cc' = token$

$updateAttribute : SYN.present' = \emptyset$

This policy is applicable only when $SYN.condition = false$, which implies $SYN.opt = \emptyset$ and $SYN.opt\_done = false$ since no primitive actions have been performed.

For a command with k parameters, there are at most $2k + 3$ policies in this phase.

As a summary, for the execution of a single command in $\mathcal{T}$ with a set of actual objects, the same set of objects are used throughout all these five phases. Specifically, in Phase I, a policy is used with SYN and each object to ensure that the object is the corresponding parameter in the simulated command. Then, in Phase II, a set of policies is used to check

the condition parts of the command between two parameter objects. If all the condition parts are satisfied, the creation (if any) and enter/remove rights operations are simulated in Phase III and IV, respectively. In Phase V, the destroy operation (if any) is performed and all objects are reset by updating their attributes. The net effect of the simulation is to update attribute acl of the object (X_k) where all operations are performed in the command.

Through this 5-phase process, the command α can be simulated by $3k + 2m + n + 5$ policies at most. This shows that a SO-TAM scheme can be simulated by a UCON scheme with at most $l(3K+2M+N+5)+|RGT|$ policies, where l is the number of the commands in a SO-TAM scheme, K, M, and N are the maximum number of parameters, condition parts, and primitive operations in all commands, respectively. After the mapping of the initial state of SO-TAM to UCON$_A$, each step in SO-TAM can be simulated with these set of policies with the same set of parameters.

Hence, we conclude that a SO-TAM system can be mapped to a UCON$_A$ system with a bounded simulation, which means that a command in the SO-TAM system can be simulated with a bounded number of policies in UCON$_A$, and the same set of parameters is used in the simulation. That is, the number of the steps in the simulation is independent of the number of the objects in the system state. This proves that UCON$_A$ is at least as expressive as SO-TAM. □

From [21] we can find that SO-TAM is strongly equivalent to TAM in expressive power, which implies a bounded simulation. Therefore we can easily conclude that UCON$_A$ is at least as expressive as TAM. Furthermore, we have the following result.

Theorem 4. *UCON$_A$ is at least as expressive as ATAM.*

Proof. ATAM is similar to TAM except that it allows testing for presence of rights

in commands which TAM does not permit. The predicates in UCON$_A$ can easily test for the absence of rights such as $r \notin [s, o]$ by means of the predicate $(s, r) \notin o.acl$ in a UCON authorization rule. Therefore, we can simulate SO-ATAM by the same mechanism as introduced in the proof of Theorem 3 for the UCON$_A$ simulation of SO-TAM. We can conclude that UCON$_A$ has at least the expressive power of SO-ATAM, which has been proven to be as expressive as ATAM [21] by bounded simulation. So for any ATAM system, we can simulate it with a SO-ATAM system, which in turn, can be mapped to a UCON$_A$ system with bounded simulation. This proves that UCON$_A$ is at least as expressive as ATAM. □

Corollary 1. *UCON$_A$ is more expressive than TAM.*

Proof. It has been shown that ATAM is equivalent to TAM but only if the simulation of each ATAM command is allowed to be done by an unbounded number of TAM commands [21]. Further ATAM cannot be simulated using only a bounded number of TAM commands. Therefore we can find a system in ATAM, which can be mapped to a UCON$_A$ system with bounded simulation and cannot be mapped to a TAM system with bounded simulation. Therefore, UCON$_A$ is more expressive than TAM. □

4.3 Expressive Power of UCON$_B$

This section compares the expressive power of UCON$_A$ and UCON$_B$. First, I show that a general UCON$_A$ ($preA_3$) model can be reduced to a UCON$_B$ ($preB_3$) model. Then I show that a general UCON$_B$ ($preB_3$) model can be reduced to a UCON$_A$ ($preA_3$) model. This verifies that these two models have the same expressive power.

4.3.1 An Example

Before formally comparing the expressive power between UCON$_A$ and UCON$_B$, an intuitive example is presented here to show the basic idea of the reductions.

Suppose a facility (e.g., an office) can be used only by a faculty in the ITE school, and the faculty needs to sign a statement that the usage of this facility is for academic activities of the ITE school. In the UCON model, this is an obligation for accessing the facility. A policy can be defined with pre-authorization and pre-obligation, where the authorization predicate is the role membership requirement of ITE faculty, and the obligation action is to sign the statement, with the requesting subject as the obligation subject and the statement as the obligation object. The following policy specifies this.

$$UCON_{AB}\_access(s, o, o_b)$$
$$(s.role = ITE\_faculty) \wedge (o.statement = o_b) \wedge sign(s, o_b) \rightarrow permit(s, o, r)$$

An access of a subject to the object with this policy can be simulated with two steps in a UCON$_A$ system, where $signed$ is a subject attribute to record the statement that the subject has signed. In the first step, the subject $sign$s the statement, with an authorization policy where the predicate is trivially true, and an update to attribute $signed$. In the second step, a subject accesses the object with the attributes $role$ and $signed$. The two UCON$_A$ policies are defined as the follows.

$$UCON_A\_sign\_statement(s, o_b)$$
$$true \rightarrow permit(s, o_b, sign)$$
$$updateAttribute : s.signed' = o_b$$

$UCON_A\_access(s, o)$

$(s.role = ITE\_faculty) \wedge (s.signed = o.statement) \rightarrow permit(s, o, r)$

$updateAttribute : s.signed' = null$

In the first policy, as the result of the signing statement permission, the $signed$ attribute of the subject is updated to record the statement object. In the second policy, both the subject's $role$ and $signed$ are checked before the access, and the $signed$ is updated to the initial value after the access. This shows how the original UCON policy (a combination of UCON$_A$ and UCON$_B$) can be simulated with two UCON$_A$ policies.

For the other direction, consider the case that the model only requires the membership of the ITE faculty, that is, the policy is a pure authorization policy as in the following.

$UCON_A\_access(s, o)$

$(s.role = ITE\_faculty) \rightarrow permit(s, o, r)$

To simulate this UCON$_A$ policy with a UCON$_B$ policy, an obligation $try\_access(s, o)$ is defined where s is the requesting subject and o is the target object, and this obligation is trivially true. The policy is:

$UCON_B\_access(s, o, statement)$

$(s.role = ITE\_faculty) \wedge try\_access(s, o) \rightarrow permit(s, o, r)$

where predicate $(s.role = ITE\_faculty)$ is a predicate to identify the that obligation $try\_access(s, o)$ is required for this access, where the obligation subject s must be an ITE faculty member.

4.3.2 Reducing UCON$_A$ to UCON$_B$

Lemma 1. *A general UCON$_A$ system can be reduced to a UCON$_B$ system.*

Proof. For a UCON$_A$ system $\mathcal{U}_a$ with scheme (ATT_a, R_a, P_a, C_a), we construct a UCON$_B$ system $\mathcal{U}_b$ with scheme $(ATT_b, R_b, P_b, B_b, C_b)$, where $ATT_b = ATT_a$, $R_b = R_a \cup \{null\}$, $P_b = P_a$. There is a single obligation action try_access which is trivially true. The policies C_b are constructed as in the following steps.

First, consider a policy $c_a \in C_a$ as the following:

$c_a(s, o)$:

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \rightarrow permit(s, o, r)$

$act_1; act_2; \ldots; act_k$

where $p_1, \ldots, p_i$ are predicates built on attributes of s and/or o; $act_1, \ldots, act_k$ are primitive actions on s and/or o.

The following policy $c_b \in C_b$ is defined in the UCON$_B$ system to simulate c_a.

$c_b(s, o)$:

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \wedge try\_access(s, o) \rightarrow permit(s, o, r)$

$act_1; act_2; \ldots; act_k$

Here predicates $p_1, \ldots, p_i$ are not for authorization purpose, but to identify that a particular obligation is required, in which the obligation subject (s) and the obligation object (o) must satisfy these predicates. The actions in c_b are the same as that in c_a. Therefore with the same state t of $\mathcal{U}_a$ and $\mathcal{U}_b$, if $t \twoheadrightarrow_{c_a(s,o)} t_a$ and $t \twoheadrightarrow_{c_b(s,o)} t_b$, then $t_a = t_b$. This proves that $\mathcal{U}_a$ can be reduced to $\mathcal{U}_b$. □

4.3.3 Reducing UCON$_B$ to UCON$_A$

This reduction is more complex since a UCON$_B$ policy can have more than two parameters while a UCON$_A$ policy only takes two parameters. For a general UCON$_B$ system $\mathcal{U}_b$ with scheme $(ATT_b, R_b, P_b, B_b, C_b)$, we construct a UCON$_A$ system $\mathcal{U}_a$ with scheme (ATT_a, R_a, P_a, C_a). The construction is performed with two cases.

Lemma 2. *A general UCON$_B$ system can be reduced to a UCON$_A$ system.*

Proof. This is proved by the following two cases.

Case 1: $sb = s$ **and** $ob = o$ **for all obligations**

In this case, let $ATT_a = ATT_b \cup \{ob\_satisfied\}$ and $R_a = R_b \cup \{r_i^b | b_i \in B_b\}$, where $dom(ob\_satisfied) \subseteq O \times B$. Consider a policy $c_b \in C_b$ as the following:

$c_b(s, o)$:

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \wedge b_1(s, o) \wedge \cdots \wedge b_j(s, o) \rightarrow permit(s, o, r)$

$act_1; act_2; \ldots; act_k$

where $p_1, \ldots, p_i$ are predicates built on the attributes of s and/or o; $b_1, \ldots, b_j$ are obligation actions; $act_1, \ldots, act_k$ are primitive actions on s and/or o.

To simulate this policy in $\mathcal{U}_a$, a set of policies is defined in $\mathcal{U}_a$ as the following.

$c_{a1}(s, o)$:

$true \rightarrow permit(s, o, r_1^b)$

$updateAttribute : s.ob\_satisfied' = s.ob\_satisfied \cup (o, b_1)$

...

$c_{aj}(s, o)$:

$true \rightarrow permit(s, o, r_j^b)$

$updateAttribute : s.ob\_satisfied' = s.ob\_satisfied \cup (o, b_j)$

$c_a(s, o)$:

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \wedge ((o, b_1) \in s.ob\_satisifed) \wedge \ldots ((o, b_j) \in s.ob\_satisifed) \rightarrow$

$permit(s, o, r)$

$updateAttribute : s.ob\_satisfied' = \emptyset$

$act_1; act_2; \ldots; act_k$

Enforcing these policies with the same parameter subject and object results in the same state transition in both $\mathcal{U}_a$ and $\mathcal{U}_b$. Specifically, the first j policies enable s to perform the obligations and record the results in the subject attribute $ob\_satisfied$. The last policy captures the same predicates, permission, as well as primitive actions as in c_b. An update action resets the $ob\_satisfied$ attribute to empty for future access with this subject.

With fixed sets of obligation actions and policies in $\mathcal{U}_b$, the number of policies in $\mathcal{U}_a$ is $\mathcal{O}(|C_b| \times |B_b|)$.

Case 2: $sb \neq s$ **or** $ob \neq o$ **in any obligation policy**

If in a policy in $\mathcal{U}_b$, an obligation subject is not the subject requesting access, or an obligation object is not the accessing target object, the reduction is different since a UCON$_A$ policy only has two parameters. We use an approach similar to reducing SO-TAM to UCON$_A$. Note that a predicate is built on attributes of at most two objects.

First, a subject SYN is introduced in $\mathcal{U}_a$. Similar to case 1, $R_a = R_b \cup \{r_i^b | b_i \in$

$B_b\}$. $ATT_a = ATT_b \cup \{cp, param, predicate, obligation, condition, update\}$, where the semantics and value domains are as follows.

- $cp \in \{token\} \cup \{c_b^n | c_b \in C_b, 0 \leq n \leq M\}$, where M is the maximum number of parameters in all the polices in C_b. The value of this attribute is the current policy in which an object is a parameter. An object whose cp has value c_b^n is involved in c_b as the nth parameter. For $n = 0$, c_b^0 is a value used only for SYN. The value $token$ specifies that no policy is being enforced with this object.

- $param \subseteq \{po_n\}$, $1 \leq n \leq K$. This attribute records the parameter objects in a simulation. $po_i \in SYN.param$ indicates that the ith parameter object is present.

- $predicate \subseteq \{pred_n\}$, $1 \leq n \leq M$, where M is the maximum number of predicates in all the policies in C_b. This attribute records the predicate information in a policy. We have $pred_n \in SYN.predicate$ iff the nth predicate in an obligation policy is true.

- $obligation \subseteq \{oblig_n\}$, $1 \leq n \leq J$, where J is the maximum number of obligations in all the policies in C_b. This attribute records the satisfaction of obligations.

- $condition \in \{true, false\}$: attribute to specify whether all predicates and obligations are satisfied in a policy.

- $update \in \{true, false\}$: attribute to specify whether all the update actions have been performed in an obligation policy.

Consider a general UCON$_B$ policy as the following.

$c_b(o_1, o_2, \ldots, o_m)$:

$p_1(o_{sp1}, o_{op1}) \wedge p_2(o_{sp2}, o_{op2}) \wedge \cdots \wedge p_i(o_{spi}, o_{opi}) \wedge b_1(o_{sb1}, o_{ob1}) \wedge b_2(o_{sb2}, o_{ob2}) \cdots \wedge$

$b_j(o_{sbj}, o_{obj}) \rightarrow permit(o_1, o_2, r)$

$[createObject\ o_2]$

up_1; up_2; $\ldots$; up_k;

$[destoryObject\ o_1]$;

$[destoryObject\ o_2]$;

where $sp1, op1, \ldots, spi, opi$ and $sb1, ob1, \ldots, sbj, obj$ are integers from 1 to m. Note that by convention, o_1 is the accessing subject, and o_2 is the target object. If c_b a creating policy, o_2 is the child object, and, without loss of generality, we assume that the first action is the creation, and the last two actions are destroy actions if c_b includes these. The simulation has five phases.

Phase I: This phase initializes a simulation of c_b, starting by the following policy with SYN.

$c_a\_phaseI\_SYN$ (SYN, SYN):

$(SYN.cp = token) \rightarrow permit(SYN, SYN, null)$

$updateAttribute : SYN.cp' = c_b^0$

This policy checks SYN's cp attribute: if the value is $token$, the execution of a new policy can be started, and this attribute's value is updated to the policy name.

We need a set of policies to set the corresponding attributes to the involved objects in a policy. Specifically, for a parameter o_n in c_b, where $1 \leq n \leq m$.

$c_a\_phaseI\_o_n$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (po_n \notin SYN.param) \wedge (o.cp = token) \rightarrow permit(SYN, o, null)$

$updateAttribute : o.cp' = c_b^n$

$updateAttribute : SYN.param' = SYN.param \cup \{po_n\}$

This policy updates the cp attribute of o to c_b^n to prevent interference from other policies. At the same time, the value c_b^n indicates that o is the nth parameter in c_b. Also the update of $SYN.param$ prevents taking more than one parameter object of o_n at a time.

Note that if c_b is a creating policy, then o_2 is the child object, and policy $c_a\_phaseI\_o_2$ is not included in the simulation. For a policy with m parameters, there are $m + 1$ policies in this phase if there is no creating operation in this policy; otherwise, there are m policies since there is at most one creating action in c_b.

Phase II: This phase includes a set of policies to check the predicates and obligations in c_b. Specifically, for a predicate $p_n(o_{spn}, o_{opn})$ where $1 \leq n \leq i$ and $1 \leq spn, opn \leq m$, we define two policies: one to update the o_{spn}'s $predicate$ attribute, and another to update SYN's attribute based on this change.

$c_a\_phaseII\_p_n\_o_{spn}\_o_{opn}$ (o_1, o_2):

$(o_1.cp = c_b^{spn}) \wedge (o_2.cp = c_b^{opn}) \wedge p_n(o_1, o_2) \rightarrow permit(o_1, o_2, null)$

$updateAttribute : o_1.predicate' = o_1.predicate \cup \{pred_n\}$

$c_a\_phaseII\_p_n\_SYN\_o_{spn}$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^{spn}) \wedge (pred_n \in o.predicate) \rightarrow permit(SYN, o, null)$

$updateAttribute : SYN.predicate' = SYN.predicate \cup \{pred_n\}$

For an obligation action $b_n(o_{sbn}, o_{obn})$ required by c_b, where $1 \leq n \leq j$, an authorization policy is defined to allow the obligation action; there are also two policies to update attributes to capture the obligation satisfaction, as shown in the following.

$c_a\_phaseII\_b_n\_o_{sbn}\_o_{obn}$ (o_1, o_2):

$(o_1.cp = c_b^{sbn}) \wedge (o_2.cp = c_b^{obn}) \rightarrow permit(o_1, o_2, r_n^b)$

$updateAttribute : o_1.obligation' = o_1.obligation \cup \{oblig_n\}$

$c_a\_phaseII\_b_n\_SYN\_o_{sbn}$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^{sbn}) \wedge (oblig_n \in o.obligation) \rightarrow permit(SYN, o, null)$

$updateAttribute : SYN.obligation' = SYN.obligation \cup \{oblig_n\}$

As c_b has i predicates and j obligations, there are $2(i + j)$ policies in this phase.

Phase III: After the simulation in Phase II, all the satisfied predicates and obligations have been recorded in SYN's attributes. In this phase there is one policy to create the object o_2 (if c_b is a creating policy) and update its attributes.

If c_b is a creating policy, the creation is the first action in the body. The following simulates it.

$c_a\_phaseIII\_SYN\_o_2\_create$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (pred_1 \in SYN.predicate) \wedge (pred_2 \in SYN.predicate) \wedge$

$\cdots \wedge (pred_i \in SYN.predicate) \wedge$

$(oblig_i \in SYN.obligation) \wedge (oblig_2 \in SYN.obligation) \wedge \cdots \wedge (oblig_j \in$

$SYN.obligation) \rightarrow permit(SYN, o, create)$

$createObject\ o$

$updateAttribute : o.cp' = c_b^2$

$updateAttribute : o.condition' = true$

$updateAttribute : SYN.condition' = true$

$updateAttribute : SYN.param' = SYN.param \cup \{po_2\}$

This policy first creates a new object as the parameter o_2 in c_b. At the same time, the $condition$ attribute of o_2 and SYN is assigned to be true since all the conditions parts have been satisfied.

If c_b is not a creating policy, the following policy just updates SYN and o_2's $condition$ value based on the evaluations in Phase II.

$c_a\_phaseIII\_SYN\_o_2$ **(**SYN, o**):**

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^2) \wedge (pred_1 \in SYN.predicate) \wedge (pred_2 \in SYN.predicate) \wedge \cdots \wedge (pred_i \in SYN.predicate) \wedge$

$(oblig_1 \in SYN.obligation) \wedge (oblig_2 \in SYN.obligation) \wedge \cdots \wedge (oblig_j \in SYN.obligation) \rightarrow permit(SYN, o, null)$

$updateAttribute : SYN.condition' = true$

$updateAttribute : o.condition' = true$

Phase IV: After Phase III, o_2's attribute has been updated according to the conditions in c_b. In this phase, update actions are simulated with the following policy.

$c_a\_phaseIV\_o_1\_o_2$ **(**o_1, o_2**):**

$(o_1.cp = c_b^1) \wedge (o_2.cp = c_b^2) \wedge (o_2.condition = true) \rightarrow permit(o_1, o_2, r)$

$up_1, \ldots, up_k$;

$updateAttribute\ o_2.update' = true$;

The last update indicates that all updates in c_b are performed. Note that as an obligation policy, c_b only can update o_1's and o_2's attributes. So only one policy is needed.

Phase V: After all the updates have been completed, we need to reset the attributes of all the subjects and objects involved in this policy for other policies. At first we need a policy to notice that all the operations, except destroy, have been completed.

c_a_$phaseV$_SYN_o_2 (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^2) \wedge (o.update = true) \rightarrow permit(SYN, o, null)$

$updateAttribute : SYN.update' = true$

If c_b has one destroy action, then it is the last action in the body. If c_b has two destroy actions, then they are the last two actions in the body. The following policies are needed.

c_a_$phaseV$_SYN_o_1_$destroy$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^1) \wedge (SYN.update = true) \rightarrow permit(SYN, o, destroy)$

$destroyObject\ o$

$updateAttribute\ SYN.param' = SYN.param - \{po_1\};$

c_a_$phaseV$_SYN_o_2_$destroy$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^2) \wedge (SYN.update = true) \rightarrow permit(SYN, o, destroy)$

$destroyObject\ o$

$updateAttribute\ SYN.param' = SYN.param - \{po_2\};$

After this step, all the primitive operations have been completed. We need to reset all the subjects' and objects' attributes. Similar to the policies in Phase I, a set of policies between SYN and the parameter objects are defined as follows, where $1 \leq n \leq m$.

c_a_$phaseV$_SYN_o_n_$Success$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (o.cp = c_b^n) \wedge (SYN.update = true) \rightarrow permit(SYN, o, null)$

$updateAttribute : o.cp' = token$

$updateAttribute : o.predicate' = \emptyset$

$updateAttribute : o.obligation' = \emptyset$

$updateAttribute : o.condition' = false$

$updateAttribute : o.update' = false$

$updateAttribute\ SYN.param' = SYN.param - \{po_n\}$;

This policy checks if all the update actions in c_b have been performed. From Phase IV, this occurs only if $SYN.condition$ is true, that means, all predicates and obligations are satisfied, and all update actions have been finished. Note that if o_1 or o_2 is destroyed in c_b, then the corresponding resetting policy is not presented here. After all the objects' attributes have been reset, we need to reset the SYN's attributes.

c_a-$phaseV\_SYN\_Success$ (SYN, SYN):

$(SYN.cp = c_b^0) \wedge (SYN.update = true) \wedge (|SYN.param| = 0) \rightarrow permit(SYN, SYN, null)$

$updateAttribute : SYN.cp' = token$

$updateAttribute : SYN.prdicate' = \emptyset$

$updateAttribute : SYN.obligation' = \emptyset$

$updateAttribute : SYN.condition' = false$

$updateAttribute : SYN.update' = false$

After this policy, the system is ready to execute another policy.

A resetting process may be also needed if any conditions part is not satisfied with the parameters; i.e., in Phase II, if predicate p_n or obligation b_n is not satisfied, then $pred_n \notin SYN.predicate$ or $oblig_n \notin SYN.obligation$ and the simulation cannot go further. In this case, either a new parameter object can be captured, or all parameters and SYN can be reset for rollback. The following policy can reset an object so that another object can be captured as the parameter, where $1 \leq n \leq m$.

c_a_$phaseV$_SYN_o_n_$Rollback$ (SYN, o):

$(SYN.cp = c_b^0) \wedge (SYN.condition = false) \wedge (o.cp = c_b^n) \rightarrow permit(SYN, o, null)$

$updateAttribute : o.cp' = token$

$updateAttribute : o.predicate' = \emptyset$

$updateAttribute : o.obligation' = \emptyset$

$updateAttribute : SYN.param' = SYN.param - \{po_n\}$

This policy states that if $SYN.condition$ is false, then the attributes of o_n can be reset, and po_n removed from $SYN.param$ so that another object can be captured as parameter X_i. The value of $SYN.condition$ is false if either some policies in Phase II have not been executed, or at least one of the condition part (predicate or obligation) in Phase II has failed so that the condition evaluation policy in Phase III is not executed. For the first case, the simulation of c_b can be restarted after resettings. For the second case, the condition of c_b is not satisfied with the object parameters. So these objects can be reset and other objects can be involved with the policies in Phase I. Note that since this policy requires $SYN.condition = false$, then $o.condition = false$ and $o.update = false$.

Finally, the following policy is needed for the rollback of SYN

α_$phaseV$_SYN_$Rollback$ (SYN, SYN):

$(SYN.cc = \alpha_0) \wedge (SYN.condition = false) \wedge (|SYN.param| = 0) \rightarrow$

$permit(SYN, SYN, null)$

$updateAttribute : SYN.cp' = token$

$updateAttribute : SYN.predicate' = \emptyset$

$updateAttribute : SYN.obligation' = \emptyset$

For a policy with m parameters, there are at most $2m + 3$ policies in this phase.

From these five phases, a general UCON$_B$ policy can be simulated with a linear number of UCON$_A$ policies, based on the number of parameters, the number of predicates, and the number of obligation actions. This proves that a UCON$_B$ model can be reduced to a UCON$_A$ model. □

With Lemma 1 and 2, the following theorem can be derived.

Theorem 5. *UCON$_A$ and UCON$_B$ have the same expressive power.*

Proof. This is a derivation from Lemma 1 and 2. □

4.4 Discussion

As mentioned in Section 4.1, the UCON$_A$ model we have studied till now is a $preA_3$ model, since we consider the predicates in a policy as pre-authorizations and the updates as post-updates. According to the definition of the model, we can also consider the updates as pre-updates or ongoing updates. As the simulation in this chapter is built on the overall effect of a usage process, it does not depend on when the updates are performed during a usage process. Therefore, all the expressive power results of $preA_3$ are valid for $preA_1$ and $preA_2$ models.

Corollary 2. *UCON $preA_1$ and $preA_2$ are at least as expressive as ATAM, respectively.*

Proof. A simulation process similar to that of $preA_3$ can be used to prove this corollary.

Note that this result does not show any relative expressive power between $preA_1$, $preA_2$, and $preA_3$. This is a topic for future work.

Similarly, according to Theorem 5, a pre-authorization model has the same expressive power of a corresponding pre-obligation model, as the follows shows.

Corollary 3. *UCON $preA_1$ and $preB_1$ have the same expressive power, and $preA_2$ and $preB_2$ have the same expressive power.*

Proof. A simulation process similar to that between $preA_3$ and $preB_3$ can be used to prove this corollary.

Again, this does not imply any relative expressive power between $preB_1$, $preB_2$, and $preB_3$.

In ongoing authorization models, the authorizations are checked repeatedly during the usage process, and updates can be performed before, during, or after the usage. For a command of TAM, the condition (the presences of rights in cells) is checked before all primitive operations are performed, and there may be some primitive operations that can make the condition false, i.e., by deleting rights from cells; that is, a condition may not be true during the usage. Therefore the simulation presented in this chapter cannot be used to compare the expressive power between UCON ongoing authorization models and TAM. The same holds for ATAM, since adding a right into a cell in an ATAM command can make the condition false during the access. The study of expressive power for ongoing authorization and obligation models is a topic for future work.

4.5 Related Work

Expressive power and safety analysis are conflicting objectives in an access control system since HRU [23]. HRU has good expressive power but the safety problem is undecidable in general. Sandhu's Schematic Protection Model (SPM) has sufficient expressive power to simulate many protection models, while sustaining efficient safety analysis [45]. SPM introduces the notion of strong security type for subjects and objects: each subject and

object is associated with a security type when created, and this type does not change after creation. The principle behind the strong type is that all instances of a type are treated as having similar behaviors.

For expressive power, SPM is less expressive than the monotonic HRU [44]. Ammann and Sandhu extended SPM to ESPM (Extended SPM) [5] by allowing multi-parent creations. The ESPM model is equivalent in expressive power to the monotonic HRU, while retaining the positive safety properties of SPM [5]. Later, Ammann, Lipton, and Sandhu [4] generalized that result that single-parent creation is less expressive than multi-parent creation in monotonic models, while it has equivalent expressive power in non-monotonic models.

Both SPM and ESPM are monotonic models, and they are not as expressive as non-monotonic HRU models. Sandhu [46] introduced TAM which generalizes HRU by introducing strong-typed subjects and objects. Both the HRU and TAM models only check the presence of a right r in the matrix cell $[s, o]$, where s is the row representing a subject and o is the column representing an object. Motivated by separation of duty policies, Ammann and Sandhu [6] extended TAM to the ATAM to allow checking the absence of rights in a cell. Ganta [21] has shown that TAM is weakly equivalent to ATAM in expressive power, but not strongly equivalent. Weakly equivalent means that a model can simulate another model possibly using an unbounded number of operations in the simulation to simulate a single operation in the simulated model, while strongly equivalence means that the simulation only requires a bounded number of operations. Some variations of TAM and ATAM have been studied in [21]. Specifically, SO-TAM is strongly equivalent to TAM in expressive power, and single-object ATAM (SO-ATAM) is strongly equivalent to ATAM. Instead

of using simulation between different models, Tripunitara and Li [57] map each access control model to a state-transition system and compare the expressive power by preserving some security properties, such as safety. As a result of this approach, TAM is shown to be less expressive than ATAM.

4.6 Summary

In this chapter I have investigated the expressive power of UCON, which is a long-standing fundamental problem in access control systems. With a formal model of UCON$_A$ and UCON$_B$, I first illustrate the expressive power with an iTunes application. Then I simulate SO-TAM with UCON$_A$, which demonstrates its ability to simulate TAM and ATAM. This formally proves that UCON$_A$ is at least as expressive as ATAM and more expressive than TAM. Furthermore, I prove that a general UCON$_A$ model can be reduced to a UCON$_B$ model, and vice versa. This formally shows that UCON$_A$ and UCON$_B$ have the same expressive power.

Chapter 5: Safety Analysis

Along with expressive power, safety is another fundamental problem in an access control model. This chapter first shows that the safety problem in general UCON$_A$ and UCON$_B$ is undecidable. Then a decidable model is obtained by imposing some restrictions, and its expressive power is studied by simulating an RBAC model and a DRM application. Without explicitly mentioning, the UCON$_A$ model in this chapter is $preA_3$, as defined in Chapter 4, and the results are also valid for $preA_1$ and $preA_2$. The safety problem of ongoing authorization models and obligation models is discussed at the end of this chapter.

Similar to the expressive power problem, for UCON condition models, as the system state changes caused by environmental information are not captured in UCON core models, safety is a function of the system environment.

5.1 Undecidability of Safety in UCON$_A$

In a UCON system, the safety question asks that, from an initial state of the system, whether or not a subject can obtain a permission on an object after a sequence of enforced policies, i.e., by updating attributes and creating/destroying objects. As the safety of a general TAM model is undecidable, and we have already shown in Chapter 4 that a general TAM can be polynomially reduced to a UCON$_A$ model, it follows that the safety of UCON$_A$ is undecidable. For the sake of completeness, we prove this with a direct reduction to the Halting

problem of a Turing machine. A construction similar to the undecidability proof of HRU [23] is used.

Theorem 6. *The safety problem of a UCON$_A$ system is undecidable.*

Proof. We show that a general Turing machine with one-directional single tape [55] can be simulated with a UCON$_A$ system, in which a particular permission leakage corresponds to the accept state of the Turing machine. A Turing machine $\mathcal{M}$ is a 7-tuple: $\{Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$, where:

- Q is a finite set of states,
- Σ is a finite set, the input alphabet not containing the special $blank$ symbol,
- Γ is a finite set, the tape alphabet, with $blank \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0, q_{accept}, q_{reject} \in Q$ are the start state, accept state, and reject state, respectively, where $q_{accept} \neq q_{reject}$.

Initially, $\mathcal{M}$ is in the state q_0. Each cell on the tape holds $blank$. The movement of $\mathcal{M}$ is determined by δ: if $\delta(q, x) = (p, y, L)$, $\mathcal{M}$ is in the state q with the tape head scanning a cell holding x, the head writes y on this cell, moves one cell to the left on the tape, and $\mathcal{M}$ enters the state p. If the head is at the left end, there is no movement. Similarly for $\delta(q, x) = (p, y, R)$, but the head moves one cell to the right.

We construct a UCON$_A$ system to simulate a Turing machine $\mathcal{M}$ introduced above, where the set of objects in a state of the UCON$_A$ system is used to simulate the cells in the tape of $\mathcal{M}$. The UCON$_A$ scheme is (ATT, R, P, C), where $R = Q \cup \{moveleft, moveright, create\}$

and $ATT = \{state, cell, parent, end\}$. For an object, the value of $state$ is either $null$ or the state of $\mathcal{M}$ if its head is positioned on this cell, the value of $cell$ is the content in the cell that the head is scanning, the $parent$ attribute stores an object identity, and end is a boolean value to show whether the head is on the right most cell of the part of the tape used so far. The set of predicates P and policies C are shown in the simulation process.

The initial state (O_0, σ_0) of the UCON$_A$ system includes a single object o_0 and its attribute assignments:

- $o_0.state = q_0$
- $o_0.cell = blank$
- $o_0.parent = null$
- $o_0.end = true$

For the state transition $\delta(q, x) = (p, y, L)$, the following policy is defined to simulate it:

$policy\_moveleft(o_1, o_2)$:

$(o_2.parent = o_1) \wedge (o_2.state = q) \wedge (o_2.cell = x) \rightarrow permit(o_1, o_2, moveleft)$

$updateAttribute : o_2.state' = null$;

$updateAttribute : o_2.cell' = y$;

$updateAttribute : o_1.state' = p$;

In this policy, the two objects are connected by the $parent$ attribute. When the Turing machine is in q_0, since o_0's $parent$ value is $null$, the left movement cannot happen. In a state when the Turing machine's state is q and the cell contains x, the left movement is simulated with a policy with parameters o_1 and o_2, where o_2's $parent$ value is o_1, and the policy updates their attributes to simulate the movement.

If the head is not scanning the right most cell, the state transition $\delta(q, x) = (p, y, R)$ can be simulated with the $policy\_moveright$, which is similar to the $policy\_moveleft$; otherwise it is simulated with the $policy\_create$, in which a new object is created.

$policy\_moveright(o_1, o_2)$:

$(o_1.end = false) \wedge (o_2.parent = o_1) \wedge (o_1.state = q) \wedge (o_1.cell = x) \rightarrow$

$permit(o_1, o_2, moveright)$

$updateAttribute : o_1.state' = null$;

$updateAttribute : o_1.cell' = y$;

$updateAttribute : o_2.state' = p$;

$policy\_create(o_1, o_2)$:

$(o_1.end = true) \wedge (o_1.state = q) \wedge (o_1.cell = x) \rightarrow permit(o_1, o_2, create)$

$updateAttribute : o_1.state' = null$;

$updateAttribute : o_1.cell' = y$;

$updateAttribute : o_1.end' = false$;

$createSubject\ o_2$;

$updateAttribute : o_2.parent' = o_1$;

$updateAttribute : o_2.state' = p$;

$updateAttribute : o_2.end' = true$;

$updateAttribute : o_2.cell' = blank$;

In a particular state of the UCON$_A$ system, only one of the three rights ($moveleft$, $moveright$, and $create$) is authorized according to one of the above policies, since the $state$ attribute is non-$null$ only for one object. Each policy assigns a non-$null$ value to an

object's $state$, and sets another one to $null$. The attribute end is true only for one object. Therefore, this UCON$_A$ system with these policies can simulate the operations of $\mathcal{M}$.

We need another policy to authorize a particular permission depending on the $state$ attribute of an object.

$policy\_q(o_1, o_2)$:

$(o_1.state = q_f) \rightarrow permit(o_1, o_2, q_f)$

For a Turing machine, it is undecidable to check if the state q_f can be reached from the initial state. Therefore, with the scheme of UCON$_A$, the granting of the permission q_f of a subject to an object is also undecidable. This completes the undecidability proof. □

From Chapter 4 it is known that UCON$_B$ has the same expressive power as UCON$_B$ model shown by bounded simulation. This implies the following result.

Corollary 4. *The safety problem of a general UCON$_B$ model is undecidable.*

Proof. From Lemma 1, for any UCON$_A$ system, there is a UCON$_B$ system that can simulate it polynomially in the number of policies, policy parameters, and predicates. Therefore in above proof, we can use a UCON$_B$ system to simulate the UCON$_A$ system, which in turn, can simulate the Turing machine. This derives the safety undecidability of UCON$_B$. □

5.2 Safety Decidable UCON$_A$ Model

Since the safety in a general model is undecidable, in this section we study the safety property of UCON$_A$ models by imposing some restrictions. First we prove that a model with finite attribute domains and without creating policies is safety decidable. Then we relax this

model by allowing restricted creating policies and obtain a more general decidable model. Finally we illustrate the expressive power of these restricted but decidable models.

5.2.1 Safety Analysis of UCON$_A$ without Creation

In a UCON$_A$ system, if the value domain for each attribute is finite, then each object has a finite number of attribute-value assignments. Furthermore, if the system does not have any creating policies, then the set of all possible objects in a system state is also finite and fixed, and therefore the total number of possible states of the system is finite, and the safety problem can be checked in the finite set of system states. This leads to the following result.

Theorem 7. *The safety problem of a UCON$_A$ system is decidable if:*

1. *the value domain of each attribute is finite, and*

2. *there is no creating policy in the scheme.*

Proof. The total number of states of a system is finite and bounded a-priori since there are no new created objects and each object has only a finite number of attribute-value assignments. The safety problem is reduced to the reachability problem of a finite state machine, which is decidable.

Consider a system specified by a scheme (ATT, R, P, C) and an initial state $t_0 = (O_0, \sigma_0)$. We consider the safety check of a permission (s, o, r) in the following analysis.

A system state t is characterized by a set of attribute assignments $\{o.a = v | o \in O, a \in ATT, v \in dom(a) \cup \{null\}\}$, where $O \subseteq O_0$. (Note that destroy actions are allowed, hence O is a subset of O_0.) Since O_0 is finite, and all the domains for the attributes in ATT are

finite, the set Q of all possible states of the system is finite. With this state set, we construct a deterministic finite automaton $\mathcal{FA} = (Q, \Sigma, \delta, q_0, Q_f)$ to show that the safety problem is decidable. The $\mathcal{FA}$ consists of:

- the finite set of states $Q = \{t | t = (O, \sigma), O \subseteq O_0\}$.
- the alphabet $\Sigma = C \times O_o \times O_o$.
- the transition function $\delta : Q \times \Sigma \rightarrow Q$.
- the start state $q_0 = t_o$.
- the accept states $Q_f = \{t | r \in \rho_t(s, o)\}$, a subset of states in which (s, o, r) is authorized by a policy with the corresponding attribute values of s and o.

The state transition function in $\mathcal{FA}$ can be constructed through the following algorithm:

1. For a state $t = (O, \sigma)$, an object pair (o_1, o_2), and a policy c, if $o_1 \in O$ and $o_2 \in O$, and the all the predicates in c are true with the attribute-value assignments of o_1 and o_2 in t (that means, the permission in c is authorized in this state), and all the conditions of the actions in c are satisfied, do the following:
   (a) Perform all the actions in c, if there are any. Define a state transition from t to t' with input $c(o_1, o_2)$ if $t \twoheadrightarrow_{c(o_1,o_2)} t'$. That is, t' is the state derived from t by enforcing the policy c with objects o_1 and o_2 as parameters. If the update actions do not change the attribute values (i.e., the new value in an update action is the same as the old value) and there is no destroy action, define a state transition from t to itself with input $c(o_1, o_2)$.

(b) If the body of c is empty, define a state transition from t to itself with input $c(o_1, o_2)$.

2. If any one of the predicates in c is not true with the attribute-value assignments of o_1 and o_2 in t, define a state transition from t to itself with input $c(o_1, o_2)$.

3. If $o_1 \notin O$ or $o_2 \notin O$ (i.e., o_1 or o_2 is destroyed in previous states), define a state transition from t to itself with input $c(o_1, o_2)$.

4. Repeat above steps in the initial state and every derived state of the system with every policy and every possible pair of objects in the initial state.

This algorithm terminates since there is only a finite number of states, policies, and pairs of objects. Through this algorithm, all the state transitions and accept states in $\mathcal{FA}$ have been defined. The accept states are those that authorize the permission (s, o, r).

By the construction, for each history $t_0 \rightsquigarrow t$ of the UCON$_A$ system, there is an input, the sequence of instantiated non-creating policies in $t_0 \rightsquigarrow t$, with which the $\mathcal{FA}$ moves from the initial state t_0 to t. Also, for each state reachable from the initial state in $\mathcal{FA}$, we can construct a history of the UCON$_A$ system from the initial state to this state by using the policies and object pairs in each transition step. Therefore $\mathcal{FA}$ can simulate any history of the UCON$_A$ system.

It is a known fact that the problem of determining whether an accept state can be reached is decidable in a finite automaton. This proves that the safety problem in the UCON$_A$ system is decidable. □

Corollary 5. *The complexity of safety analysis for a UCON$_A$ system without creating policies and with a finite domain of each attribute is polynomial in the number of possible*

states in the system.

Proof. Consider the finite automaton in Theorem 7 as a directed graph. The safety check for a permission (s, o, r) is to find a path from the initial state to an accept state, which is called as the PATH problem. It is known that the PATH problem of a graph is polynomial in the number of nodes. That means, the complexity of the safety problem is polynomial to the size of all possible states of the system. □

From a different view, the complexity of the safety problem for a UCON$_A$ system without creating policies and with a finite domain of each attribute is NP-hard in the number of policies in the scheme. This can be proved by simulating a monotonic mono-operational HRU without creates, which has a NP-complete safety problem in the number of commands [23, 46].

Theorem 8. *The complexity of safety problem for a UCON$_A$ system with finite sets of attribute domains and without creation is NP-hard in the number of policies in the scheme.*

Proof. From Theorem 3 in Chapter 4 we already know that a SO-TAM system can be reduced to a UCON$_A$ system with bounded simulation, while SO-TAM is equivalent to TAM, which, in turn, can simulate any form of HRU model [46]. Thus, for a monotonic mono-operational HRU model without creation, it can be polynomially reduced to a UCON$_A$ system with the following features:

1. there is no creating or destroying policy;

2. the domain of an object's access control list (attribute acl) is finite since there is no creation, and all other attributes introduced in the proof of Theorem 3 have finite domains.

From Theorem 7, this UCON$_A$ model is decidable. It is known that safety in a monotonic mono-operational HRU is NP-complete. Also it has been shown in the proof of Theorem 3 that a monotonic mono-operational HRU model can be reduced to this UCON$_A$ model polynomially. These imply that the safety problem of a general decidable UCON$_A$ model in Theorem 7 is NP-hard, since it subsumes the monotonic mono-operational HRU. □

This result is not conflicting with Corollary 5, since the polynomial complexity of safety analysis in Corollary 5 is in the size of all possible states of a system, which is generally exponential to the number of the objects in the initial state and the sizes of attribute domains. On the other hand, in Theorem 8 the number of the policies in the scheme is the parameter of the problem.

5.2.2 Safety Analysis of UCON$_A$ with Creation

The decidable model introduced above does not allow the creation of new objects in a system. In this section we relax this assumption and allow a restricted form of creation. Intuitively, if the subject's attribute values have to be updated in a creating policy, and there is no policy that can update this subject's attribute values to its previous values, then there is a finite number of objects that can be created in the system, and the safety is decidable by tracing all possible system states. We will see in Section 5.3 that there are examples of useful systems that meet this requirement. We keep the assumption of finite value domain for each attribute.

Definition 14. *An* attribute-value assignment tuple *(or simply* attribute tuple*) is a function* $\tau : ATT \rightarrow \bigcup_{a \in ATT} dom(a) \cup \{null\}$ *that assigns a value or* $null$ *to each attribute in*

ATT, *where* $\tau(a) \in dom(a) \cup \{null\}$.

For a system with a finite domain for each attribute, there is only a finite set of attribute tuples, which is denoted as $\mathcal{ATP}$. In any system state $t = (O_t, \sigma_t)$, for each object $o \in O_t$, its attribute tuple τ_o in state t is the attribute-value assignments of o in this state. Specifically, $\forall a \in ATT, \tau_o(a) = \sigma_t(o.a)$, where $\tau_o \in \mathcal{ATP}$.

Example 24 Consider a UCON$_A$ scheme with $ATT = \{a, b\}$, and $dom(a) = dom(b) = \{0, 1\}$. The $\mathcal{ATP}$ of this scheme is $\{(a = null, b = null), (a = null, b = 0), (a = null, b = 1), (a = 0, b = null), (a = 0, b = 0), (a = 0, b = 1), (a = 1, b = null), (a = 1, b = 0), (a = 1, b = 1)\}$. In each state of the system, an object's attribute-value assignment is indicated by one of $\mathcal{ATP}$. □

Grounding of Policies

For safety analysis, we generate a set of *ground* policies with a *grounding* process, for each policy in a UCON$_A$ scheme. Intuitively, grounding a policy is to evaluate the policy with all possible attribute tuples of the object parameters, and only those satisfying the predicates in the policy are considered in the safety analysis.

Consider the following generic UCON$_A$ policy

$c(s, o)$:

$p_1 \wedge p_2 \wedge \cdots \wedge p_i \rightarrow permit(s, o, r)$

$[createObject\ o]$;

$up_1; \ldots; up_m$;

$up_{m+1}; \ldots; up_n$;

$[destroyObject\ o]$;

$[destroyObject\ s]$;

where the $createObject$ and $destroyObject$ actions are optional, and $p_1, \ldots, p_i$ are predicates on s's and o's attributes. If c is a creating policy, these predicates are only based on s's attributes. Without loss of generality, we assume that $up_1, \ldots, up_m$ are update actions on o's attributes, and $up_{m+1}, \ldots, up_n$ are update actions on s's attributes, and for any attribute of an object there is at most one update in the policy. In a real command, any of the actions can be optional. For example, for a command that includes a $destroyObject\ o$ action, all update actions on o can be removed since they have no effect on the new system state.

The grounding process works as follows. For any two attribute tuples $\tau_s, \tau_o \in \mathcal{ATP}$, if all the predicates $p_1, \ldots, p_i$ are true with s's attribute tuple τ_s and o's attribute tuple τ_o, then a ground policy $c(s : \tau_s, o : \tau_o)$ is generated with the following format [1]:

$c(s : \tau_s, o : \tau_o)$:

$true \rightarrow permit(s, o, r)$

$[createObject\ o]$;

$updateAttributeTuple\ o : \tau_o \rightarrow \tau_o'$;

$updateAttributeTuple\ s : \tau_s \rightarrow \tau_s'$;

$[destroyObject\ o]$;

$[destroyObject\ s]$;

[1]Note that a different policy format is used in a ground policy from the original UCON$_A$ policy. Specifically, in a ground policy, each object parameter is associated with an attribute tuple, and all updates of attributes are reduced to two $updateAttributeTuple$ actions on the parameter objects. Therefore a single UCON$_A$ policy can generate at most $|\mathcal{ATP}| \times |\mathcal{ATP}|$ ground policies. A ground policy $c(s : \tau_s, o : \tau_o)$ can be executed only when the subject's attribute tuple is τ_s and the object's attribute tuple is τ_o.

where τ'_o is the attribute tuple of o after the update actions $up_1, \ldots, up_m$, and τ'_s is the attribute tuple of s after the update actions $up_{m+1}, \ldots, up_n$. If c is a creating policy, the predicates $p_1, \ldots, p_i$ are evaluated with τ_s only, and we can consider $\tau_o(a) = null$ for all $a \in ATT$.

This process is repeated with every possible attribute tuple τ_s and τ_o. Since each object has a finite number of attribute tuples, for any policy this grounding process is guaranteed to terminate, and a finite number of ground policies is generated. The set of ground policies is denoted as C_n.

With this grounding process, the predicate evaluation in each policy is pre-processed by considering all possible attribute tuples in a system. This simplifies the subsequent safety analysis.

Example 25 This example illustrates the grounding process for a policy and does not necessarily have a practical interpretation. For simplicity let $ATT = \{a\}$ and $dom(a) = \{1, 2, 3\}$. The policy

$c(s, o)$:

$(s.a > o.a) \rightarrow permit(s, o, r)$

$updateAttribute : o.a' = o.a + 1$;

generates the following three policies in the grounding process.

$c(s : (a = 2), o : (a = 1))$

$true \rightarrow permit(s, o, r)$;

$updateAttributeTuple\ o : (a = 1) \rightarrow (a = 2)$;

$c(s : (a = 3), o : (a = 1))$

$true \rightarrow permit(s, o, r)$;

$updateAttributeTuple\ o : (a = 1) \rightarrow (a = 2)$;

$c(s : (a = 3), o : (a = 2))$

$true \rightarrow permit(s, o, r)$;

$updateAttributeTuple\ o : (a = 2) \rightarrow (a = 3)$;

For two attribute tuples τ_s and τ_o as attribute-value assignments of s and o respectively, if $s.a > o.a$ is not true (e.g., $s.a = 1, o.a = 2$), no ground policy is generated. Here by definition we assume that the predicate $s.a > o.a$ is false if either $s.a = null$ or $o.a = null$.

□

Our goal is to use the finite set of ground policies to study the safety property of a UCON$_A$ system. With the following result, the change of the system state caused by enforcing an original policy can be simulated by enforcing a ground policy.

Lemma 3. *Given two states $t = (O, \sigma)$ and $t' = (O', \sigma')$ in a UCON$_A$ system,*

1. *if $t \twoheadrightarrow_{c(s,o)} t'$, where $c \in C$, then there is a ground policy c_n generated from c such that $t \twoheadrightarrow_{c_n(s:\tau_s, o:\tau_o)} t'$, where $\tau_s, \tau_o \in \mathcal{ATP}$.*

2. *if $t \twoheadrightarrow_{c_n(s:\tau_s, o:\tau_o)} t'$, where $c_n \in C_n$, then there is a policy $c \in C$ such that $t \twoheadrightarrow_{c(s,o)} t'$, where $\tau_s, \tau_o \in \mathcal{ATP}$.*

Proof. For the first case, let $\tau_s(a) = \sigma(s.a)$ and $\tau_o(a) = \sigma(o.a)$ for each $a \in ATT$. Since $t \twoheadrightarrow_{c(s,o)} t'$, all the predicates in c are satisfied with s and o's attribute values in the state t. According to the grounding process, trivially $c_n(s : \tau_s, o : \tau_o)$ is a valid ground policy generated from c. Also based on the grounding process, for a primitive action in c, if it is not an update action, then it is included in c_n; if it is an update action $updateAttribute : s.a = v'$, where $a \in ATT$, $v' \in dom(a)$, then $updateAttributeTuple : \tau_s \rightarrow \tau_s'$ is included

in $c_n(s:\tau_s, o:\tau_o)$, and $\tau'_s(a) = v'$. Therefore with the actions in $c_n(s:\tau_s, o:\tau_o)$, the system state changes to the same state as with $c(s,o)$.

In the second case, suppose $t \twoheadrightarrow_{c_n(s:\tau_s,o:\tau_o)} t'$, where $c_n \in C_n$. Since c_n can be enforced in t, the attribute-value assignments of s and o are τ_s and τ_o in t, respectively. According to the grounding process, this implies that all the predicates in the policy c, from which c_n is generated, are satisfied by these assignments. Therefore the policy c can be applied in t. Also, both c and c_n have the same non-update actions, and all the update actions in c have the same effect with the $updateAttributeTuple$ action(s) in c_n, hence $t \twoheadrightarrow_{c(s,o)} t'$. □

This lemma shows that from the same system state, a single step by enforcing a policy can be simulated with a single step with a ground policy, and vice versa. The following shows that a history of the system with the original policies can be simulated by a history with ground policies.

Lemma 4. *For a UCON$_A$ system with initial state t_0,*

1. *if $t_0 \rightsquigarrow_C t$, then there is a transition history $t_0 \rightsquigarrow_{C_n} t$.*

2. *if $t_0 \rightsquigarrow_{C_n} t$, then there is a transition history $t_0 \rightsquigarrow_C t$.*

Proof. The first case can be proved by induction on the number of steps in $t_0 \rightsquigarrow_C t$.

Basis step: Suppose $t_0 \twoheadrightarrow_{c(s,o)} t$, where $c \in C$. According to Lemma 3, there is a ground policy $c_n \in C_n$ such that $t_0 \twoheadrightarrow_{c_n(s:\tau_s,o:\tau_o)} t$.

Induction step: Assume that for every history $t_0 \rightsquigarrow_C t'$ with k steps, there is a history $t_0 \rightsquigarrow_{C_n} t'$. Consider a history $t_0 \rightsquigarrow_C t$ of length $k+1$ and let $t' \twoheadrightarrow_{c(s,o)} t$ be the $(k+1)$th step. Since c can be enforced in t', according to Lemma 3, there is a ground policy $c_n \in C_n$ such that $t' \twoheadrightarrow_{c_n(s:\tau_s,o:\tau_o)} t$. By induction hypothesis, there exists a history $t_0 \rightsquigarrow_{C_n} t$. This

completes the induction step and the proof of the first case. A similar approach can be used for the proof of the second case. □

With this lemma, we can conclude that for a UCON$_A$ system, the set of all states reachable from the initial state using the original policies can be reached using the ground policies, and vice verse. Therefore we can study the safety property of the system with the set of ground policies.

Attribute Creation Graph

The basic idea of our safety analysis is to allow a finite number of creating steps from any subject in the initial state. This requires that in a creating ground policy, the child's attribute tuple must be different from the parent's attribute tuple, so that if the creating relation is acyclic, there only can be finite steps of creating from the original subject.

Definition 15. *A ground policy is a* creating ground policy *if it contains a* $createObject$ *action in its body; otherwise, it is a* non-creating ground policy.

Definition 16. *In a creating ground policy* $c_n(s : \tau_s, o : \tau_o)$*,* τ_s *is the* create-parent attribute tuple*, and* τ'_o *is the* create-child attribute tuple.

This definition implicitly requires that in each creating ground policy, the child's attribute tuple is updated. Without loss of generality, we assume that if there is no update action for the child in a creating policy, then $\tau_o = \tau'_o$ in all the ground policies generated from this creating policy; that is, they are both null-valued attribute assignments.

Definition 17. *The* generation value *of an object* o *is defined recursively as follows:*

1. *if* $o \in O_0$*, its generation value is 0;*

2. *if o is created by a creating ground policy $c(s : \tau_s, o : \tau_o)$, its generation value is one more than the generation value of s.*

Definition 18. *For a UCON$_A$ system with finite attribute domains, the* attribute creation graph (ACG) *is a directed graph with nodes all the possible attribute tuples $\mathcal{ATP}$, and an edge from τ_u to τ_v if there is a creating ground policy in which τ_u is the create-parent attribute tuple and τ_v is the create-child attribute tuple.*

Lemma 5. *In a UCON$_A$ system, if the ACG is acyclic and in each creating ground policy the child's attribute tuple is updated, then the set of all possible generation values is finite, and the maximal generation value is $|\mathcal{ATP}|$.*

Proof. With an acyclic ACG, in each creating ground policy the create-child attribute tuple is different from the create-parent attribute tuple, otherwise there is a self-loop with this attribute tuple and the ACG is not acyclic. If the maximal generation value is more than $|\mathcal{ATP}|$, then there exist creating ground policy $c_1(s_1 : \tau_{s1}, o_1 : \tau_{o1})$ and $c_2(s_2 : \tau_{s2}, o_2 : \tau_{o2})$, where τ_{o1} is τ_{s2} or an ancestor of τ_{s2} and τ_{o2} is τ_{s1} or an ancestor of τ_{s1} in ACG, which is in conflict with the acyclic ACG property of the system. Therefore the set of all possible generation values is finite, and the maximal generation value is $|\mathcal{ATP}|$. □

Attribute Update Graph

As a subject can create an object, which in turn can create another object, an acyclic ACG ensures that the "depth" of these creation chains is bounded. At the same time, a subject can have unbounded number of direct children, which allows the system to have an arbitrary large number of objects. With some restrictions on the attribute update relation, our system allows only a finite number of creations with a single subject as parent. Specifically, if the

subject's attribute tuple has to be updated in a creating policy, and there is no policy in the scheme that can update the subject's attribute tuple to a previous one, then the number of the subject's direct children is finite.

Definition 19. *In a ground policy* $c_n(s : \tau_s, o : \tau_o)$*,*

- *if there is an* $updateAttributeTuple\ s : \tau_s \rightarrow \tau_s'$ *action, then* τ_s *is an* update-parent attribute tuple*, and* τ_s' *is an* update-child attribute tuple.
- *if there is an* $updateAttributeTuple\ o : \tau_o \rightarrow \tau_o'$ *action, then* τ_o *is an* update-parent attribute tuple*, and* τ_o' *is an* update-child attribute tuple.

Note that in a creating ground policy in which s is the parent and τ_s is updated, τ_s is both a create-parent attribute tuple and an update-parent attribute tuple.

Definition 20. *For a UCON$_A$ system with finite attribute domains, the* attribute update graph (AUG) *is a directed graph with nodes all possible attribute tuples* $\mathcal{ATP}$*, and an edge from* τ_u *to* τ_v *if there is a ground policy in which* τ_u *is an update-parent attribute tuple and* τ_v *is an update-child attribute tuple.*

Lemma 6. *In a UCON$_A$ system, if the AUG has no cycle containing a create-parent attribute tuple, and in each creating ground policy the parent's attribute tuple is updated, then the number of children of a subject is finite, and the maximal number of children is* $|\mathcal{ATP}|$.

Proof. Since AUG has no cycle containing a create-parent attribute tuple, then in any creating ground policy $c_n(s : \tau_s, o : \tau_o)$, τ_s' is different from τ_s, otherwise there is a self-loop on the create-parent attribute tuple since in a creating ground policy, τ_s is both a

create-parent attribute tuple and an update-parent tuple. If the number of creating ground policies which can use the same subject as the parent is more than $|\mathcal{ATP}|$, then there are at least two creating policies in which the update-parent attribute tuple are the same. That means, there is a policy that updates the subject's attribute tuple to this create-parent tuple, which implies a cycle which contains this create-parent attribute tuple. This is in conflict with the property of AUG in the system. Therefore the set of all possible creating ground polices that can use this subject as parent is finite, and the maximal number of its children is $|\mathcal{ATP}|$. □

Safety Analysis

Consider a system which satisfies the requirements in Lemma 5 and 6. For a subject in the initial state of the system, the number of direct children of this subject is finite, and the creation "depth" from this subject is also finite. These two aspects ensure that in the system there is a bounded number of objects that can be created, and the safety can be checked with the finite states of the system.

Definition 21. *A* descendant *of an object is defined recursively as either itself or a child of a descendant of this object.*

Theorem 9. *The safety problem of a UCON$_A$ system with finite attribute domains is decidable if:*

- *the ACG is acyclic, and*
- *the AUG has no cycle containing a create-parent attribute tuple, and*

- *in each creating ground policy $c(s : \tau_s, o : \tau_o)$, both the parent's and the child's attribute tuples are updated.*

Proof. We first prove that the set of all possible objects that can be created in the system is finite. Consider a subject $s \in O_0$. If there are any creating ground policies that can be applied with s as parent, then, according to Lemma 6, the number of creating polices with s as parent is finite, and the maximal number of children created with s is $|\mathcal{ATP}|$. On the other hand, according to Lemma 5, for each object there is only a finite number of generation values, therefore the number of descendants of s is finite. Since the set of objects in the initial state is finite, and each object created in the system is a descendant of an object in the initial state, then there is only a finite number of objects that can be created in the system.

The safety analysis needs to check if a particular permission (s, o, r) can be authorized in any reachable state of the system. For this purpose we use the recursive algorithm shown in Figure 5.1 to search a state that enables the permission (s, o, r) in all the states of the system reachable from the initial state. The algorithm starts from the initial state of the system, and checks all reachable states with the non-creating ground policies. If there is no state where the permission is enabled, from every state of the reachable states, the algorithm generates a new object and recursively does the similar check. This step is repeated with all possible sequences of creations until all reachable states are checked.

First we prove that this algorithm terminates. Since in each call of $SafetyCheck()$, there are finitely many reachable states, and each state has a finite number of objects, then the number of loops in each call is finite. According to the properties of the systems, the set of all objects that can be created is finite, hence the number of calling $SafetyCheck()$

Safety Check Algorithm
// input: UCON$_A$ system with initial state $t_0 = (O_0, \sigma_0)$ and a finite set of ground policies
1) $SafetyCheck(O_0, t_0)$
2) Construct a finite state automaton $\mathcal{FA}$ with objects O_0 and the set of non-creating ground policies. (refer to the proof in Theorem 7.)
3) **foreach** $t_0 \rightsquigarrow t$ in $\mathcal{FA}$ **do**
4) **if** $r \in \rho_t(s, o)$, **return** *true*
5) **foreach** $t_0 \rightsquigarrow t$ in $\mathcal{FA}$, where $t = (O, \sigma)$, **do**
6) **foreach** subject s in t **do**
7) **foreach** creating ground policy $c(s : \tau_s, o : \tau_o)$, where $\tau_s(a) = \sigma(s.a)$ **do**
8) enforce $c(s : \tau_s, o : \tau_o)$;
9) create object o and update its attribute tuple to τ'_o;
10) update s's attribute tuple to τ'_s;
11) the system state changes to t' with new object o and updated attributes of s and o;
12) $SafetyCheck(O_0 \cup \{o\}, t')$;
13) **return** *false*

Figure 5.1: Safety check algorithm

is finite. Therefore the algorithm terminates in a finite number of steps.

Then we show that all the reachable states of the system are visited by this algorithm if the permission (s, o, r) is not enabled in any state. In each call of $SafetyCheck()$, all possible states without creating new objects are checked in the first loop (line 3-4). For a particular subject and a particular creating ground policy, the policy can be applied with the subject at most once because the AUG has no cycle containing any create-parent attribute tuple. In line 7 every possible creating policy is applied for a subject as parent at least once. So in the loops of 5-6 all possible sequences of creating policies are applied, and the reachable states with created objects are also visited until no object can be created. Therefore the algorithm checks all the possible reachable states in the system.

So if a state is reached where the permission (s, o, r) is enabled according to a policy,

the algorithm returns *true*. By checking all possible non-creating policy sequences (line 2-4) for reachable states and trying all possible sequence of creating policies in each reachable state, if the algorithm reaches a state in which the permission (s, o, r) is enabled, then there is a sequence of policies leading the system from the initial state to this state. This proves that this algorithm can perform the safety analysis. □

From Lemma 6 and 5, it is known that the maximum number of all possible descendants of an object is $|\mathcal{ATP}| \times |\mathcal{ATP}|$. For a UCON$_A$ system with initial state $t_0 = (O_0, \sigma_0)$, the maximum number of all possible created objects is $|O_0| \times |\mathcal{ATP}|^2$. On the other hand, for each object, the maximum number of its attribute-value assignments is $|\mathcal{ATP}|$. According to the safety check algorithm, the maximum number of steps ($SafetyCheck$) is

$$(|O_0| \times |\mathcal{ATP}|) * ((|O_0|+1) \times |\mathcal{ATP}|) * ((|O_0|+2) \times |\mathcal{ATP}|) * \cdots * ((|O_0| + N) \times |\mathcal{ATP}|),$$

where $N = |O_0| \times |\mathcal{ATP}|^2$. Therefore the complexity of this safety check algorithm is $\mathcal{O}\big(((|O_0| + N) \times |\mathcal{ATP}|)^N\big)$. On the other hand, according to Theorem 8, the safety problem is NP-hard on the number of policies in the scheme, since it subsumes the model without creation shown in Theorem 7.

5.3 Expressive Power of Decidable UCON$_A$ Models

Certain restricted UCON$_A$ models have decidable safety, so the question does arise whether or not these models can capture practically useful access control policies. In this section we use these limited forms of decidable UCON$_A$ models to express practically useful policies that have been discussed in the literature. We show that UCON$_A$ without creation can

simulate an RBAC96 model with URA97 administrative scheme, and that UCON$_A$ with restricted creation can express policies for a DRM application with consumable rights. These examples demonstrate that our decidable models maintain practical expressive power.

5.3.1 RBAC Systems

In an RBAC system, a subject can be viewed as having a role attribute whose value is a subset of the roles in the system. Similarly, an object can have a role attribute for each right indicating the subset of roles for which that right is authorized. In classic RBAC [19, 50] these role attributes are fixed and changeable only by administrative actions, which could themselves be authorized based on roles. Thus possession of a suitable administrator role would enable a subject to change the roles of other subjects and objects, essentially accomplishing the user-role assignment and permission-role assignment which are the basic operations of administrative RBAC (ARBAC). In this section we consider the user-role assignment (URA97) portion of the ARBAC97 model [49] and express it with a decidable UCON$_A$ system.

An RBAC scheme consists of a set of regular roles RR and a partial order relation $\geq_{RH} \subseteq RR \times RR$ for the role hierarchy, a set of administrative roles AR and a partial order relation $\geq_{ARH} \subseteq AR \times AR$ for the administrative role hierarchy, a fixed set of generic rights RT, and a set of rules to change user-role assignments, embodied in the can_assign and can_revoke relations of URA97 [49]. An RBAC system state consists of a set of subjects SUB, a set of permissions PER, a set of user-role assignments $UA \subseteq SUB \times RR$, a set of user-administrative role assignments $UAA \subseteq SUB \times AR$, and a set of permission-role assignments $PA \subseteq PER \times RR$. The permissions are defined by objects and rights,

$PER \subseteq OBJ \times RT$, where OBJ is a set of objects. Note that here we simply consider a user in the original RBAC as a subject in UCON$_A$ and do not account for role activation explicitly. The construction can be easily extended to do this.

For each RBAC system, we construct a UCON$_A$ system with scheme (ATT, R, P, C), where $ATT = \{ua, uaa, acl\}$, ua and uaa are subject attributes to store the user-role assignments and user-administrative role assignments in RBAC, respectively, and acl is an object attribute to record the permission-role assignments. $R = RT \cup \{assign\_r | r \in RR\} \cup \{revoke\_r | r \in RR\}$. The set of predicates P consists of:

- the predicate $x \in y$ to indicate that x is an element of set y;
- the predicate $member$ to check if a role or any of its senior roles is assigned to a subject, and $member(r, s.ua) = true$ if $\exists r' \geq_{RH} r, r' \in s.ua$;
- the predicate $notmember$ to check that a role or all of its senior roles is not assigned to a subject, and $notmember(r, s.ua) = true$ if $\forall r' \geq_{RH} r, r' \notin s.ua$;
- the predicate $admin\_member$ checks if an administrative role or any of its senior roles is assigned to a subject, and $admin\_member(r, s.uaa) = true$ if $\exists r' \geq_{ARH} r, r' \in s.uaa$.

With fixed $\geq_{RH}$ and $\geq_{ARH}$ relations, all these predicates are polynomially computable.

The initial state of the RBAC system $(SUB_0, OBJ_0, PER_0, UA_0, UAA_0, PA_0)$ is mapped to a UCON$_A$ state (O_0, σ_0), where $O_0 = SUB_0 \cup OBJ_0$ and σ_0 as a set of attribute-value assignments shown below.

- $s_0.ua = \{r | r \in RR, and\ (s, r) \in UA_0\}$ for $s_o \in SUB_0$;

- $s_0.uaa = \{r | r \in AR, and\ (s, r) \in UAA_0\}$ for $s_o \in SUB_0$;
- $o_0.acl = \{(r, rt) | r \in RR, rt \in RT, (o_0, rt) \in PER_0, and\ (r, (o_0, rt)) \in PA_0\}$ for $o_o \in OBJ_0$;

The set of policies C is defined as follows. First, a set of policies is needed to specify the original permissions of RBAC in a state of the UCON$_A$ system. For a role $r \in RR$ and a right $rt \in RT$, the policy is shown below.

$policy\_r\_rt(s, o)$:

$member(r, s.ua) \wedge ((r, rt) \in o.acl) \rightarrow permit(s, o, rt)$

Note that roles and rights are not parameters in a policy. With the RBAC scheme, the upper bound on the number of these policies is $|RR| \times |RT|$ in the simulating UCON$_A$ scheme.

In URA97, a relation can_assign specifies which particular administrative role can assign a subject, which satisfies a prerequisite condition, to a role in a specified role range. A prerequisite condition is a boolean expression generated by the grammar $cr :\equiv x | \bar{x} | cr \wedge cr | cr \vee cr$, where $x \in RR$. For a subject $s \in SUB$ in a state, x is true if $\exists x' \geq_{RH} x, (s, x') \in UA$ and $\bar{x}$ is true if $\forall x' \geq_{RH} x, (s, x') \notin UA$. The set of the prerequisite conditions in an RBAC is denoted as CR. Therefore $can\_assign \subseteq AR \times CR \times 2^{RR}$.

Consider the rule $can\_assign1(ar, cr, [r_1, r_2])$, where $ar \in AR$, $cr = x \wedge \bar{y}$, $x, y \in RR$. It can be expressed by a bounded set of policies in UCON$_A$, one for each $r_i \in [r_1, r_2]$:

$can\_assign\_r_i(s_1, s_2)$:

$admin\_member(ar, s_1.uaa) \wedge member(x, s_2.ua) \wedge notmember(y, s_2.ua) \rightarrow$

$permit(s_1, s_2, assign\_r_i)$

$updateAttribute : s_2.ua' = s_2.ua \cup \{r_i\}$

This policy allows a subject s_1 to assign the role r_i ($r_i \in [r_1, r_2]$) to the subject s_2 when s_1 is a member of the administrative role ar, and s_2 is a member of the role x but not of y. The number of policies to simulate $can\_assign1$ is bounded, since for fixed RR and $\geq_{RH}$, the number of roles in $[r_1, r_2]$ is bounded.

Similarly, a revocation relation in URA97 can be expressed with policies in UCON$_A$. A $can\_revoke \subseteq AR \times 2^{RR}$ relation specifies that a subject with membership in an administrative role can revoke a subject's membership in the role r if r is in a particular role range. This implies that r is assigned to the subject before the revocation. We can simulate $can\_revoke1(ar, [r_1, r_2])$ with a set of policies, one for each role $r_i \in [r_1, r_2]$:

$can\_revoke\_r_i(s_1, s_2)$:

$admin\_member(ar, s_1.uaa) \wedge (r_i \in s_2.ua) \rightarrow permit(s_1, s_2, revoke\_r_i)$

$updateAttribute : s_2.ua' = s_2.ua - \{r_i\}$

This policy states that in a particular state, a subject s_1 can execute the right $revoke\_r_i$ on the subject s_2 by removing r_i ($r_i \in [r_1, r_2]$) from s_2's ua attribute, if ar or one of its seniors is in the s_1's uaa and r_i is in the subject s_2's ua. Again, the number of policies to simulate $can\_revoke1$ is bounded since the number of roles in $[r_1, r_2]$ is bounded for fixed RR, $\geq_{RH}$, AR, and $\geq_{ARH}$.

This shows that a UCON$_A$ system can be constructed to simulate an RBAC system with URA97 administrative scheme. In this UCON$_A$ system, each attribute's value domain is finite since RR, AR, and RT are all fixed sets, and there is no creating policy in the

system. According to Theorem 7, this UCON$_A$ system has decidable safety, which implies this RBAC system also has decidable safety.

Based on the same processes, we can simulate an RBAC system with PRA97 (permission-role assignment model in ARBAC97) using UCON$_A$ and show that this RBAC model also has decidable safety. For an RBAC system with RRA97 (role-role assignment model in ARBAC97), since RR and $\geq_{RH}$ are not fixed, this approach cannot be used to prove the decidability of its safety problem.

5.3.2 DRM applications with Consumable Rights

Consumable access is becoming an important aspect in many applications, especially in DRM. For example, in a pay-per-use application, a user's credit is reduced after an access to an object, causing the user to lose the right on the object after a number of accesses. For another example, if an object can only be accessed by a fixed number of subjects concurrently, a subject's access may revoke the access right of another subject. Most applications with consumable rights can be modelled by UCON with the mutability property [39, 40].

Consider a DRM application, where a user can order a music CD, along with a license file which specifies that the CD can only be copied a fixed number of times (say, 10). The license file can be embedded with the CD or distributed separately, and must be available and respected by the CD copying software or device. A subject (user) has an attribute $credit$ with a numerical value of the user's balance. Each object (CD) has an attribute $copylicense$ to specify how many copies that a subject can make with this object. The policies are defined as follows.

$order(s, o)$:

$(s.credit \geq o.price) \wedge (o.owner = null) \rightarrow permit(s, o, order)$

$updateAttribute : s.credit' = s.credit - o.price$

$updateAttribute : o.owner' = s$

$updateAttribute : o.copylicense' = 10$

$allow\_copy(s, o)$:

$(o.owner = s) \wedge (o.copylicense > 0) \rightarrow permit(s, o, allowcopy)$

$updateAttribute : o.allowcopy' = true$

$copy(o_1, o_2)$:

$(o_1.allowcopy = true) \rightarrow permit(o_1, o_2, copy)$

$createObject\ o_2$

$updateAttribute : o_2.sn' = o_1.copylicense$

$updateAttribute : o_1.copylicense' = o_1.copylicense - 1$

$updateAttribute : o_1.allowcopy' = false$

The first policy specifies that a user can order an object if not ordered before (the value of attribute $owner$ is $null$) and the user's credit is larger than the object's price. As a result of the order, the user's credit is reduced, the object's $owner$ is updated to the user's ID, and the object's $copylicense$ is set to 10. The second policy states that whenever the object's $copylicense$ is positive, the owner of the object is allowed to make a copy of the object. In the third policy, if an object is allowed to be copied, a new object (CD) can be created, its sn (serial number) is set to be the original object's $copylicense$ value, and the original object's $copylicense$ is reduced by one. As the newly created object does not have any

license information, it cannot be copied.

In a system with a fixed number of users and objects in the initial state, the value domain of $owner$ is finite since there are no new users can be created. The set of all possible values for $credit$ of a subject is finite, since the value is set after pre-payment or registration. Note that the changes of the $credit$ value because of administrative actions, e.g., credit card payment, are not captured in the model. The value domains for $copylicense$ and $allowcopy$ are obviously finite. Therefore, all the attribute value domains are finite sets. Furthermore, there is only one creating policy, in which both the child's and the parent's attributes are updated, and there is no cycle with any create-parent attribute tuples since the value of $copylicense$ strictly decreases. According to Theorem 9, the safety of this UCON$_A$ model is decidable.

In this example we focus on the policy definition with UCON$_A$ model. How to ensure the availability and trusted update of a license file are implementation issues and not included here.

5.4 Discussion

The UCON$_A$ model we have studied till now in this chapter is $preA_3$. Similar to the expressive power problem, all the undecidable and decidable results in this chapter are valid for $preA_1$ and $preA_2$ models.

For ongoing authorization core models, as an attribute update in a usage process can revoke another ongoing usage process, and the attribute updates for a revoked usage may be different from that for an ended usage by the subject, so the system state change after an access is nondeterministic. We leave the safety analysis of ongoing authorization models

for future work.

As an obligation policy can take more than two parameters and check attribute predicates between them, the safety decidable results we have achieved may not be valid. We leave this for future work.

5.5 Related Work

Previous work in safety analysis has shown that, for some general access control models such as HRU [23], safety is an undecidable problem. That means, there is no algorithm to determine that, given a general access control matrix system, whether it is possible to find a combination of commands to produce a state where a subject has a particular permission. HRU did provide decidability results for special cases with either mono-operational commands (only one primitive operation allowed in a command) or mono-conditional (only one presence check in the condition part of a command) monotonic (no “destroy subject” or “destroy object” or “remove right” operations) commands. These restricted models are very limited in expressive power. The take-grant model has a linear time algorithm to check the safety property, but it also has limited expressive power [14, 32].

Sandhu [46] introduces the TAM model which generalizes the HRU by introducing strong-typed subjects and objects. The monotonic form of TAM with acyclic scheme is decidable, and the decision procedure is NP-hard. Extending TAM, Soshi [56] presents a dynamic-typed access matrix model (DTAM), which allows the type of an object to change dynamically within a fixed domain. The decidable model of DTAM allows non-monotonic operations.

Motwani et al. [34] present an accessibility decidable model in a capability-based system, which is a generalized take-grant model and a restricted form of HRU. The approach to the safety problem is based on its relationship to the membership problem in languages generated by certain classes of string grammars. Jaeger and Tidswell [24] provide a safety analysis approach which uses a basic set of constraints on a system. More recently, Koch et al. [29] report on results that use a graph transformation model to specify access control policies. The state is represented by a typed labelled graph and state transitions by graph transformations. Under some restrictions on the form of the rules (e.g., rules that add or delete elements), the model has a decidable accessibility problem, and the rules can model restrictive forms of DAC and a simplified decentralized RBAC. Very recently, Li and Tripunitara [31] use a trust management approach to study the safety problem in RBAC and derive the decidability of safety with a user-role administration scheme (URA97). The first safety decidable model obtained in this chapter has the capability to simulate an RBAC system with URA97.

5.6 Summary

In this chapter I investigate the safety property of UCON. First I show that the safety problem in general UCON$_A$ models is shown to be undecidable by simulating a Turing machine. This also shows that the safety problem of UCON$_B$ is undecidable in general. Then I prove that a UCON$_A$ model with finite attribute domains and without creating policies is decidable, and the safety problem is NP-hard. Further, by relaxing the creation restriction I prove that the safety problem is decidable for a UCON$_A$ model with acyclic attribute creation

graph and no cycles that include create-parent tuple in attribute update graph. The decidable models are shown to be useful by simulating RBAC96 model with URA97 scheme, and a DRM application with consumable rights.

Chapter 6: Conclusions and Future Work

6.1 Conclusions

Based on the conceptual model presented in previous work, a temporal logic model of UCON is proposed in this dissertation. In this model, authorizations are specified as predicates on subject and object attributes, obligations are specified as subject actions, and conditions are specified as predicates on system attributes. A UCON policy is a set of instantiated logical rules, where the set of scheme rules has the properties of soundness and completeness. The specification flexibility of this logic model is shown by expressing policies for various applications.

With a policy-based model formalizing the overall effect of a usage process, the expressive power of UCON has been formally studied. First I show that $preA$ is more expressive than TAM, and at least as expressive as ATAM. Then, by defining a simulation relation, I show that $preA$ and $preB$ have the same expressive power.

On the safety aspect, I show that a general $preA$ model is safety undecidable, which implies the same result for a general $preB$ model. Furthermore, a restricted $preA$ model with finite attribute domains and acyclic creation relation based on attribute values has a decidable safety property, and the complexity of the safety problem is NP-hard. This restricted model maintains good expressive power, as shown by simulating a practical RBAC model with user-role assignment scheme, and a DRM application with consumable rights.

6.2 Future Work

This dissertation lays the groundwork for considerable future work on UCON. First of all, an administrative model of UCON should be developed, including attribute management and administrative policies. As UCON is an attribute-based model, synchronized attribute acquisition and management are required in a system and should be included in an administrative model. Also mentioned in Chapter 2, post-obligations are in the scope of an administrative model. For example, if a subject does not satisfy an obligation after an access, a security administrator needs to take compensatory actions according to administrative policies.

Secondly, practically useful and efficiently decidable cases of UCON based on current results should be investigated in the future. As shown in Chapter 5, the safety problem of the restricted UCON$_A$ is NP-hard. In practice a tractable safety property is desired. We conjecture that not only object creation, but also attribute update in UCON affect the safety problem. Restrictions on update actions can be a direction to find better safety results.

An important property related to ongoing decision checks in UCON is concurrency. As a subject can have multiple ongoing accesses, or an object can be accessed by multiple subjects simultaneously, a subject or an object attribute can be a shared variable of theses concurrent accesses. This implies that an ongoing update in one access can affect the status of another access, such as revocation. Concurrency affects safety analysis in two aspects. On one side, as specified in Chapter 3, an update after a $revokeaccess$ action (e.g., due to updates in a concurrent access) may be different from an update after an $endaccess$ action, e.g., by updating different attributes, or updating the same attribute but with different values. This introduces nondeterminism for safety analysis. On the other

side, the sequence of update actions in concurrent accesses affects state transitions in a system. Specifically, concurrent accesses can lead a system to a different state from that with serialized accesses. This introduces another kind of nondeterminism. While capturing the essential aspect of state transitions and permission leakages caused by the mutability of UCON, in this work we use a simplified approach for the safety analysis with ongoing authorizations and obligations. The safety problem in concurrent environments is a topic for future work.

This dissertation focuses on the policy and model layers of UCON in the OM-AM framework. As mentioned in Section 2.1, a UCON model can be supported by different architectures and implemented by several mechanisms. Emerging trusted computing hardware such as Trusted Computing Group (TCG) [3] and LaGrande Technology (LT) [1], and trusted operating systems such as Microsoft Next-Generation Secure Computing Base (NGSCB) [2], provide new mechanisms to implement UCON policies and motivate new architectures in real systems [52]. For example, the traditional server-side reference monitor can be complemented with client-side control and audit with client-side reference monitor [42, 52]. Emerging applications of UCON with these new architectures and mechanisms will be studied in the future.

Bibliography

Bibliography

[1] LaGrande technology for safer computing. http://www.intel.com/technology/security.

[2] Next-generation secure computing base. http://www.microsoft.com/resources/ngscb.

[3] *TCG Specification Architecture Overview*. https://www.trustedcomputinggroup.org.

[4] P. Ammann, R. Lipton, and R. Sandhu. The expressive power of multi-parent creation in monotonic access control models. In *Proceedings of the Computer Security Foundation Workshop*, 1992.

[5] P. Ammann and R. Sandhu. Safety analysis for the extended schematic protection model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[6] P. Ammann and R. Sandhu. Implementing transaction control expressions by checking for absence of access rights. In *Proceedings of the Annual Computer Security Applications Conference*, 1992.

[7] D. E. Bell and L. J. Lapadula. Secure computer systems: Mathematical foundations and model. *Mitre Corp. Report No.M74-244, Bedford, Mass.*, 1975.

[8] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A temporal access control mechanism for database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), February 1996.

[9] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transaction on Database Systems*, 23(3), September 1999.

[10] E. Bertino, C. Bettini, and P. Samarati. A temporal authorization model. In *Proceedings of ACM Conference on Computer and Communication Security*, 1994.

[11] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, 2001.

[12] C. Bettini, S. Jajodia, X. Sean Wang, and D. Wijesekera. Obligation monitoring in policy management. In *Proceedings of the 3rd Internationl Workshop on Policies for Distributed Systems and Networks*, 2002.

[13] C. Bettini, S. Jajodia, X. Sean Wang, and D. Wijesekera. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th VLDB Conference*, 2002.

[14] M. Bishop. Theft of information in the take-grant protection model. In *Proceedings of IEEE Computer Security Foundation Workshop*, 1988.

[15] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*, 1988.

[16] J. Chomicki and J. Lobo. Monitors for history-based policies. In *Proceedings of the 2nd Internationl Workshop on Policies for Distributed Systems and Networks*, 2001.

[17] N. Damianou, N. Dulay, E. Lupu, , and M. Sloman. The ponder policy specification language. In *Proceedings of the Workshop on Policies for Distributed System s and Networks*, 2001.

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

[19] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. Richard Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3), 2001.

[20] A. Gal and V. Atluri. An authorization model for temporal data. In *Proceedings of the ACM Conference on Computer and Communication Security*, 2000.

[21] S. Ganta. *Expressve Power of Access Contrtol Models Based on Propagation of Rights*. PhD thesis, George Mason University, 1996.

[22] M. Hansen and R. Sharp. Using interval logics for temporal analysis of security protocols. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*, 2003.

[23] M. H. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communication of ACM*, 19(8), 1976.

[24] T. Jaeger and J. E. Tidswell. Practical safey in flexible access control models. *ACM Transactions on Information and Systems Security*, 4(2), May 2001.

[25] S. Jajodia, P. Samarati, , and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium On Research in Security and Privacy*, 1997.

[26] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2), June 2001.

[27] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1), 2005.

[28] J. Joshi, E. Bertino, B. Shafiq, and A. Ghafoor. Constraints: Dependencies and separation of duty constraints in gtrbac. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*. ACM, 2003.

[29] M. Koch, L. V. Mancini, and F. Paris-Presicce. Decidability of safety in graph-based models for access control. In *Proceedings of the 7th European Symposium on Research in Computer Security, LNCS 2502*, 2002.

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

[31] N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Techniques*, 2004.

[32] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of ACM*, 24(3), 1977.

[33] Z. M. and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems Specification*. Springer-Verlag, 1991.

[34] R. Motwani, R. Panigrahy V. Saraswat, and S. Venkatasubramanian. On the decidability of accessibility problem (extended abstract). In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*, 2000.

[35] OASIS. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML)*.

[36] OASIS XACML TC. *Core Specification: eXtensible Access Control Markup Language (XACML)*, 2005.

[37] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and Systems Security*, 3(2), 2000.

[38] J. Park. *Usage Control: A Unified Framework for Next Generation Access Control*. PhD thesis, George Mason University, 2003.

[39] J. Park and R. Sandhu. The UCON_{ABC} usage control model. *ACM Transactions on Information and Systems Security*, 7(1), February 2004.

[40] J. Park, X. Zhang, and R. Sandhu. Arrtibute mutability in usage control. In *Proceedings of the Proceedings of 18th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, 2004.

[41] J. S. Park and R. Sandhu. RBAC on the web by smart certificates. In *Proceedings of ACM Workshop on Role-Based Access Control*, 1999.

[42] J. S. Park and R. Sandhu. Binding identities and attributes using digitally signed certificates. In *Proceedings Annual Computer Security Applications Conference*, 2000.

[43] J. S. Park, R. Sandhu, and G. Ahn. Role-based access control on the web. *ACM Transactions on Information and Systems Security*, 4(1), 2001.

[44] R. Sandhu. Expressive power of the schematic protection model. In *Proceedings of the Computer Security Foundation Workshop*, 1988.

[45] R. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of ACM*, 35(2), 1988.

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

[47] R. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11), November 1993.

[48] R. Sandhu. Engineering authority and trust in cyberspace: The OM-AM and RBAC way. In *Proceedings of Fifth ACM Workshop on Role-based Access Control*, 2000.

[49] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2, 1999.

[50] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role based access control models. *IEEE Computer*, 29(2), 1996.

[51] R. Sandhu and J. Park. 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*, 2003.

[52] R. Sandhu and X. Zhang. Peer-to-peer access control architecture using trusted computing technology. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, 2005.

[53] F. Siewe, A. Cau, and H. Zedan. Compositional framework for access control policies enforcement. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*, 2003.

[54] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *IEEE Computer Security Foundations Workshop*, 1997.

[55] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.

[56] M. Soshi. Safety analysis of the dynamic-typed access matrix model. In *Proceedings of the 6th European Symposium on Research in Computer Security, LNCS 1895*, 2000.

[57] M. V. Tripunitara and N. Li. Comparing the expressive power of access control models. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2004.

[58] X. Wang, G. Lao, T. DeMartini, H. Reddy, M. Nguyen, and E. Valenzuela. Xrml – extensible rights markup language. In *Proceedings of the 2002 ACM workshop on XML security*, 2002.

[59] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Transactions on Information and Systems Security*, 8(4), 2005.

[60] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *Proceedings of 9th ACM Symp. on Access Control Models and Tech.*, 2004.

Curriculum Vitae

Xinwen Zhang was born on January 1st, 1974, in Hunan, P. R. China and is a citizen of P. R. China. He received the Bachelor and Master of Engineering in Power Engineering from Huazhong University of Science and Technology, Wuhan, China in 1995 and 1998, respectively. During 1998-2000, he was a research student in Nanyang Technological University, Singapore, and a software development engineer of CE-Infosys Pte Ltd, Singapore. Currently he is a Ph.D. candidate in the Laboratory for Information Security Technology and the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia, USA.