Safety Analysis of Usage Control Authorization Models

Xinwen Zhang
George Mason University
xzhang6@gmu.edu

Ravi Sandhu
George Mason University and TriCipher Inc.
sandhu@gmu.edu

Francesco Parisi-Presicce
George Mason University and
Univ. di Roma La Sapienza, Italy
fparisip@gmu.edu

ABSTRACT

The usage control (UCON) model was introduced as a unified approach to capture a number of extensions for traditional access control models. While the policy specification flexibility and expressive power of this model have been studied in previous work, as a related and fundamental problem, the safety analysis of UCON has not been explored. This paper presents two fundamental safety results for UCON$_A$, a sub-model of UCON only considering authorizations. In UCON$_A$, an access control decision is based on the subject and/or the object attributes, which can be changed as the side-effects of using the access right, resulting in possible changes to future access control decisions. Hence the safety question in UCON$_A$ is all the more pressing since every access can potentially enable additional permissions due to the mutability of attributes in UCON. In this paper, first we show that the safety problem is in general undecidable. Then, we show that a restricted form of UCON$_A$ with finite attribute value domains and acyclic attribute creation relation has a decidable safety property. The decidable model maintains good expressive power as shown by specifying an RBAC system with a specific user-role assignment scheme and a DRM application with consumable rights.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

access control, usage control, UCON, authorization, safety

1. INTRODUCTION

Modern information systems require fine-grained and flexible access control policies, which need dynamic and expressive access control models. Traditional access control models, such as access matrix [5], mandatory access control (MAC) [1, 3], discretionary access control (DAC), and role-based access control (RBAC) [15, 4], have been formulated to meet different application requirements. Recently, usage control (UCON) [11] was proposed as a general and comprehensive model to extend the underlying mechanism of traditional access control models. In [11, 19], the policy specification flexibility and expressive power of UCON has been shown in access control systems, digital rights management (DRM), and trust management applications, among others.

A different but related important problem in access control is the leakage of permissions. 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 state of the system, and this, in turn, may enable other permissions. This dynamic property makes it is difficult to foresee a system state in which a subject can have a particular right on a particular object. This is referred to as the safety problem in access control. 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, if at all possible.

Access control in UCON is made by policies of authorizations, obligations, and conditions (also referred as UCON$_{\mathrm{ABC}}$ model [11]). In UCON$_A$, the control decision of an access is determined by one or more predicates built from the attributes of the subject and the object. A particularly powerful innovation of UCON$_A$ is that an access can result in the updates of the subject's and/or the object's attributes as side-effects. These updates, in turn, may result in the changes of the permissions of future accesses. The resultant permission propagations, because of attribute mutability, make the safety analysis complex and untractable in general UCON models.

This paper presents two main contributions to the safety analysis of UCON$_A$. First, we prove that the safety problem in general UCON$_A$ is undecidable by reduction to the halting problem in Turing machines. Second, two decidable models of UCON$_A$ are obtained with some restrictions in the general model. Specifically, the safety problem is decidable for a UCON$_A$ model with finite attribute domains and without "creating" policies. Also, the safety problem is decidable for a UCON$_A$ model with finite attribute domains and "creating" policies, where the attribute creation graph is acyclic. We then show that these restricted forms of UCON$_A$

are practically useful by specifying policies for RBAC systems and DRM applications.

The rest of the paper is organized as follows. Section 2 contains an introduction to and a formal definition of UCON$_A$. Section 3 presents the undecidability result of the safety problem in general UCON$_A$ systems. We then present two decidable models with some restrictions on the general UCON$_A$ in Section 4, and their expressive power in Section 5. Section 6 gives some related work on safety analysis in access control models. Section 7 concludes the paper and presents some further directions of research.

2. USAGE CONTROL MODEL

In this section, first we briefly review the main components of the UCON model in [11], and then we present a policy-based formal UCON$_A$ model.

2.1 A Brief Introduction

A UCON system consists of six components: subjects and their attributes, objects and their attributes, generic rights, authorizations, obligations, and conditions, where authorizations, obligations and conditions are the components of usage control decisions. An attribute is regarded as a variable with a value assigned to it in each system state. Authorizations are predicates based on subject and/or object attributes, such as role name, security classification or clearance, credit amount, etc. Obligations are actions that are performed by subjects or by the system. For example, playing a licensed music file requires a user to click an advertisement, and downloading a white paper requires a user to fill out a form. Conditions are system and environmental restrictions such as system clock, location, system load, system mode, etc.

In UCON, a complete usage process consists of three phases: before-usage, ongoing-usage, and after-usage. The control decision components are checked and enforced in the first two phases, called pre-decisions and ongoing-decisions respectively, while no decision check is defined in the after-usage phase (since there is no control after a subject finishes an access on an object). The presence of ongoing decisions is called the continuity of UCON.

Another important property of UCON is attribute mutability. Mutability means that one or more subject or object attribute values can be updated as the results of an access. Along with the three phases, there are three kinds of updates: pre-updates, ongoing updates, and post-updates. All these updates are performed and monitored by the security system. The updating of attributes as side-effect of subject activity is a significant extension of classic access control where the reference monitor mainly enforces existing permissions. Changing subject and object attributes has impact on the future usage of permissions involving this subject or object. This aspect of mutability makes UCON very powerful as discussed in [11, 19] but also makes the safety question much more important.

For each decision component (authorizations, obligations, and conditions) in UCON, a number of core models are defined based on the phase where usage control is checked and updates are performed. For example, in authorization core models, usage control decisions are dependent on subject and object attributes, which can be checked and determined before or during a usage process, and are called $preA$ (pre-authorizations) and onA (ongoing authorizations), respectively. Based on possible updates in all three phases, each sub-model has four core models. For example, $preA_0$ is the core model with pre-authorizations and without updates, and $preA_1$, $preA_2$, and $preA_3$ are core models with pre-authorizations, and pre-updates, ongoing updates, and post-updates, respectively. Similar core models have been defined for onA, $preB$ (pre-obligations), onB (ongoing obligations), $preC$ (pre-conditions), and onC (ongoing conditions). We mention obligations and conditions for completeness, but do not consider them any further in this paper.

In this paper we focus on the safety analysis of UCON $preA$ models. Since an authorization decision is determined by subject's and object's attributes, and these attribute values can be updated as side-effects of the authorization, the safety problem in authorization models is more pressing than that in obligation and condition models. For UCON onA models, the system state changes non-deterministically, depending on concurrent accesses and reasons for attribute updates (e.g., ended access vs. revoked access). We leave the safety analysis of onA models for future work. For the sake of simplicity in this paper we refer UCON$_A$ as UCON $preA$ models.

2.2 A Formal Model of UCON$_A$

A logical model of UCON is presented in [19] to capture the new features of UCON, such as the attribute mutability and the decision continuity, but it is not appropriate to study the safety problem. The main reason is that the logical model focuses on the specification of the detailed state change of the system in a single usage process, while for safety analysis, the overall effect of a usage process and the permission propagation as the cumulative result of a sequence of usage processes need to be formulated. Therefore a new formal model is developed in this paper to capture the global effect of a usage process and the cumulative result of a sequence of usage processes. Specifically, in this model, 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 don't specify precisely how the serialization is achieved, since there are many known standard techniques for this purpose. The details of how to achieve serialization is an implementation-level issue as opposed to a model-level issue. Focusing on model-level issues, we define a set of policies 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.

2.2.1 *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 be 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[1]

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.

[1]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.

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

2.2.2 Attributes, Values, and States

Each object is specified with a non-empty and finite set of attributes, such as group membership, role, security clearance, credit amount, etc, defined by the system designer. 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. Without loss of generality, we assume that in a system, each object has the same fixed set of attribute names ATT.

Each attribute name is treated as a variable of a specific datatype, which determines the attribute's domain and the set of functions that can be used with the attribute values. The domain of the attribute a is denoted as $dom(a)$, and we assume that for each $a \in ATT$, $null \notin dom(a)$.

EXAMPLE 1. *Each subject (user) in an organization has the same set of attribute names $ATT = \{adminRole, regRole\}$, where the $adminRole$'s value is an administrative role name and the $regRole$'s value is a regular role name. An administrator in the organization has no-null values for both attributes, while a regular employee's $adminRole$ is $null$, which is set when the subject is created and cannot be updated.* □

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

EXAMPLE 2. *Consider an organization in which RBAC [15] is enforced. Each subject has an attribute ua, which stores all the roles explicitly assigned to this subject by the security officer, and whose domain consists of all possible subsets of roles in the system. Another attribute dev_ua is defined to store a single role that an employee (say Alice) can be assigned to within the development department. If $R_{dev} = \{$roles within the development department$\}$, then $dom(dev\_ua) = R_{dev} \cup \{null\}$. For Bob, who is in the testing department, the dev_ua value is always $null$. A possible system state (if no other objects and attributes exist in the system) is $t = \{Alice.ua = \{p_1, p_2\}, Alice.dev\_ua = \{p_1, p_2\}, Bob.ua = \{p_3\}, Bob.dev\_ua = null\}$, where p_1, p_2, p_3 are role names, and $p_1, p_2 \in R_{dev}$.* □

2.2.3 Predicates

DEFINITION 2. *A predicate $p(s, o)$ is a boolean-valued polynomially computable function 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 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 $true$ in the current state of a system if s's $credit$ attribute value is larger than \$100 in this state. A predicate can be defined with a number of attributes from a single object or two objects. For 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 and constants, e.g., $s.cleareance \geq o.classification$, $s.credit \geq o.value$, $(s, r) \in o.acl$, where $o.acl$ is the object o's access control list. Note that the attributes in a predicate can be from a single subject or object, or one subject and one object.

2.2.4 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 3. *A* primitive action *(or simply* action*) is a state transition of a system. Three primitive actions of UCON$_A$ are defined as in the Figure 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 $updateAttribute$ 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.

Although all these primitive actions are actually performed by the system, they are the results of the accesses performed by subjects. External actions or events of a system are not directly captured in UCON$_A$. For example, in an online reading application, the decrease of a credit after a user reads a chapter is an update action captured by the UCON$_A$ model, while the increase of the user's total credit amount with a credit card payment is an external event, and is not regarded as an action in the system. To capture these external events, UCON$_A$ will need to be extended with an administrative model. The safety question investigated in this paper is therefore in absence of an explicit administrative model.

2.2.5 UCON$_A$ Policy

Satisfied predicates on attributes in UCON$_A$ 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 actions, e.g., by creating a new object, or updating attribute values. These actions, in turn, may make other predicates satisfied, and then enable other permissions and system state changes. The safety analysis of UCON$_A$ 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.

DEFINITION 4. *A policy of UCON$_A$ 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

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

Figure 1: Primitive actions in UCON$_A$

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, ..., 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.

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 state of the system. Note that there may be policies that have no actions but only authorization rules, which cause no state transitions. 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.

Instead of specifying the individual state changes in a single usage process, the policy-based formalization 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 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 safety results in this paper derived for $preA_3$ 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 paper is a $preA_3$ model.

DEFINITION 5. *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 one $createObject$ action at most. Without loss of generality, we assume that in a creating policy, the first parameter s, which is the *unique parent* object, must exist before the actions, and o is created as a *child* object.

Without loss of generality, we can also 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 in the last one. Negation is not explicitly required since we can always define a new predicate equivalent to a negated one. 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 conditions for each primitive action are satisfied, then the permission is granted, and all the primitive actions are performed. Otherwise, the permission is not granted, and the system does not change state. As we assume that all accesses are serialized, and the enforcement of each policy is atomic, either an access is granted and all primitive actions are completed, or the system state does not change.

EXAMPLE 3. *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:*

$create\_doc(s, doc)$:
$(s.role = sci) \to permit(s, doc, create)$
$createObject\ doc$
$updateAttribute$: $doc.readTimes = 10$

$read\_doc(s, doc)$:
$(s.role = anonymous) \wedge (doc.readTimes > 0) \to$
$permit(s, doc, read)$
$updateAttribute$:
$doc.readTimes = doc.readTimes - 1$

The first creating policy specifies that a subject with role of 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, $readTimes$ is decreased by one.* □

2.2.6 UCON$_A$ Protection System

A formal representation of a UCON$_A$ system can be defined with the basic components that we have introduced.

DEFINITION 6. *A UCON$_A$* 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$_A$* protection system *(or simply* system*) is specified by a UCON$_A$ scheme and an initial state (O_0, σ_0).*

DEFINITION 7. *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.*

The function ρ_t maps a pair (subject, object) to a set of generic rights, according to their attribute-value assignments in the 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 each pair (s, o) is $\mathcal{O}(|P| \times |C|)$.

DEFINITION 8. *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 $t \rightsquigarrow t'$.*

3. SAFETY UNDECIDABILITY IN UCON$_A$

In a UCON$_A$ system, the safety question asks whether or not, from an initial state of the system, a subject can obtain a permission on an object after a sequence of enforced policies, i.e., by updating attributes and creating/destroying objects. In this section we show that the safety problem for a general UCON$_A$ model is undecidable by reducing it to the halting problem of a general Turing machine.

THEOREM 1. *The safety problem of a UCON$_A$ system is undecidable.*

Proof Sketch. A general Turing machine with one-directional single tape [17] can be simulated with a UCON$_A$ system, in which a particular permission leakage corresponds to the accept state of the Turing machine. A construction similar to the undecidability proof of the access matrix model [5] is used. Specifically, the tape in a Turing machine is simulated with a set of objects, and a set of object attributes is defined to indicate the Turing machine's state, the content in each cell, and the cell that the head is scanning. A set of UCON$_A$ policies is defined to simulate the state transition function of the Turing machine. As for a Turing machine, it is undecidable to check if its accept state can be reached from the initial state. Therefore, with the scheme of simulating UCON$_A$, the granting of the particular permission of a subject to an object is also undecidable. This proves the safety undecidability of UCON$_A$. The full construction is presented in the Appendix. □

4. SAFETY DECIDABLE UCON$_A$ MODELS

Since the safety of the general model is undecidable, in this section we study the safety property of UCON$_A$ models with some restrictions. First we prove that a model with finite attribute domains and without creating policies is safety decidable. Then we relax this restriction by allowing restricted creating policies and obtain a more general decidable model. Finally we illustrate the expressive power of these decidable models.

4.1 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 2. *The safety problem of a UCON$_A$ system is decidable if:*

1. *the value domain of each attribute is finite, and*
2. *there are no creating policies in the scheme.*

Proof. The total number of states of the 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.

Let the system be 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 $\Sigma = C \times O_0 \times O_0$.
- the transition function $\delta : Q \times \Sigma \rightarrow Q$.
- the start state $q_0 = t_0$.
- the accept states $Q_f = \{t | r \in \rho_t(s, o)\}$, a (sub)set 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 a 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 or not is decidable in a finite state machine. This proves that the safety problem in the UCON$_A$ system is decidable. □

COROLLARY 1. *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 2 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. □

4.2 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 that there are examples of useful systems that meet this requirement. We keep the assumption of finite value domain for each attribute.

DEFINITION 9. *An* attribute-value assignment tuple *(or simply* attribute tuple*) is a function $\tau : ATT \rightarrow dom(ATT) \cup \{null\}$ that assigns a value or $null$ to each attribute in ATT.*

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 this state is the attribute-value assignments in this state. Specifically, $\forall a \in ATT, \sigma_t(o.a) = \tau_o(a)$, where $\tau_o \in \mathcal{ATP}$.

4.2.1 *Grounding 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:

$c(s : \tau_s, o : \tau_o)$:
$true \rightarrow permit(s, o, r)$
$[createObject\ o]$;
$updateAttributeTuple\ o : \tau_o \rightarrow \tau'_o$;
$updateAttributeTuple\ s : \tau_s \rightarrow \tau'_s$;
$[destroyObject\ o]$;
$[destroyObject\ s]$;

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 4. *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 other 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 1. *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 2. *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 1, 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 last step. Since c can be enforced in t', according to Lemma 1, 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.

4.2.2 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 10. *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 11. *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 12. *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 in a creating ground policy $c(s:\tau_s, o:\tau_o)$, its generation value is one more than the generation value of its parent s.*

DEFINITION 13. *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 3. *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 two creating ground policies, $c_1(s_1:\tau_{s1}, o_1:\tau_{o1})$ with create-child attribute tuple τ'_{o1} and $c_2(s_2:\tau_{s2}, o_2:\tau_{o2})$ with create-child attribute tuple τ'_{o2}, and 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. Therefore there is a cycle in the 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 $|\mathcal{ATP}|$. □

4.2.3 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 an 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, a system can allow 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 14. *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 15. *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 4. *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 $|\mathcal{ATP}|$. □

4.2.4 Safety Analysis

Consider a system with satisfies the requirements in Lemma 3 and 4. 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 16. *A* descendant *of an object is defined recursively as either itself or a child of a descendant of this object.*

THEOREM 3. *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 4, 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 3, 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 2 to search for 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 a 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 $SafetyCheck()$ 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 3 and 4, it is known that the maximum number of all possible descendants of an object is $|\mathcal{ATP}| \times |\mathcal{ATP}|$. For

```
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 2.)
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 $\tau'_o$;
10)             update $s$'s attribute tuple to $\tau'_s$;
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 2: Safety check algorithm

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 $\mathcal{O}\big(((|O_0| + N) \times |\mathcal{ATP}|)^N\big)$.

5. 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.

5.1 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 [15, 4] 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 [16] 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 [16]. 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 2, 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.

5.2 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 [12, 11].

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 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 3, the safety of this UCON$_A$ model is decidable.

6. RELATED WORK

Previous work in safety analysis has shown that, for some general access control models such as the access matrix model formalized by Harrison, Russo, and Ullman (HRU model) [5], safety is an undecidable problem. That means, there is no algorithm to determine, 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 [2, 9].

Sandhu's Schematic Protection Model (SPM) has sufficient expressive power to simulate many protection models, while provides efficient safety analysis [13]. 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. Sandhu [14] introduces a typed access matrix model (TAM) model which generalizes the HRU model 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 [18] 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. [10] present an accessibility decidable model in a capability-based system, which is a generalized take-grant model and a restricted form of HRU model. 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 [6] provide a safety analysis approach which uses a basic set of constraints on a system. More recently, Koch et al. [7] 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 [8] 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 paper has the capability to simulate an RBAC system with URA97.

7. CONCLUSION AND FUTURE WORK

In this paper we investigate the safety property of UCON. First the safety problem in general UCON$_A$ models is shown to be undecidable by simulating a Turing machine. Then we prove that a UCON$_A$ model with finite attribute domains and without creating policies is decidable. By relaxing the creation restriction, we also 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. These two results lay the groundwork for considerable future work on these topics, and hold out the promise for discovery of practically useful and efficiently decidable cases of UCON.

In this paper we focus only on the safety analysis with pre-authorization policies in UCON. For condition core models of UCON, as system state changes caused by environmental information are not captured in UCON core models, safety is a function of the system environment. For obligation core models, as specified in [19], an obligation of an access is an action that can be related to the subject requesting the access, or to some other subjects and, therefore, a usage policy may include more than two parameters. Analysis of the safety problem with obligations is for future work.

8. REFERENCES

[1] D. E. Bell and L. J. Lapadula, Secure Computer Systems: Mathematical Foundations and Model. Mitre Corp. Report No.M74-244, Bedford, Mass., 1975.

[2] M. Bishop, Theft of Information in the Take-Grant Protection Model, In Proc. of IEEE Computer Security Foundation Workshop, 1988.

[3] D. E. Denning, A lattice model of secure information flow, Communications of the ACM, vol. 19, no. 5, 1976.

[4] 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, Volume 4, Number 3, August 2001.

[5] M. H. Harrison, W. L. Ruzzo, and J. D. Ullman, Protection in Operating Systems, Communication of ACM, Vol 19, No. 8, 1976.

[6] T. Jaeger and J. E. Tidswell, Practical Safey in Flexible Access Control Models, ACM Transactions on Information and Systems Security, Vol. 4, No. 2, May 2001.

[7] M. Koch, L. V. Mancini, and F. Parisi-Presicce,Decidability of Safety in Graph-Based Models for Access Control, In Proc. of the 7th European Symposium on Research in Computer Security, LNCS 2502, 2002.

[8] 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.

[9] R. J. Lipton and L. Snyder, A Linear Time Algorithm for Deciding Subject Security, Journal of ACM, 24(3), 1977.

[10] R. Motwani, R. Panigrahy, V. Saraswat, and S. Venkatasubramanian, On the Decidability of Accessibility Problem (Extended Abstract), In Proc. of the 32th Annual ACM Symposium on Theory of Computing, 2000.

[11] J. Park and R. Sandhu, The UCON$_{\mathrm{ABC}}$ Usage Control Model, ACM Transactions on Information and Systems Security, Feb., 2004.

[12] J. Park, X. Zhang, and R. Sandhu, Attribute Mutability in Usage Control, In Proc. of the Annual IFIP WG 11.3 Working Conference on Data and Applications Security, 2004.

[13] R. Sandhu, The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes, Journal of ACM, 35(2), 1988.

[14] R. Sandhu, The Typed Access Matrix Model, In Proc. of the IEEE Symposium on Research in Security and Privacy, 1992.

[15] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, Role-Based Access Control Models, IEEE Computer, Volume 29, Number 2, February 1996.

[16] R. Sandhu, V. Bhamidipati, and Q. Munawer, The ARBAC97 Model for Role-Based Administration of Roles, ACM

Transactions on Information and Systems Security, Volume 2, Number, February 1999.

[17] M. Sipser, Introduction to the Theory of Computation, PWS Publishing 1997.

[18] M. Soshi, Safety Analysis of the Dynamic-Typed Access Matrix Model, In Proc. of the 6th European Symposium on Research in Computer Security, LNCS 1895, 2000.

[19] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu, A Logical Specification for Usage Control, In Proc. of the 9th ACM Symposium on Access Control Models and Technologies, 2004.

APPENDIX

The proof of Theorem 1:
We show that a general Turing machine with one-directional single tape [17] 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 our undecidability proof. $\square$