Program Synthesis in Administration of Higher-Order Permissions

Glenn Bruns
Bell Labs, Alcatel-Lucent
Glenn.Bruns@alcatel-lucent.com

Michael Huth
Imperial College, London
M.Huth@imperial.ac.uk

Kumar Avijit
Carnegie Mellon University
kavijit@cs.cmu.edu

ABSTRACT

In "administrative" access control, policy controls permissions not just on application actions, but also on actions to modify permissions, on actions to modify permissions on *those* actions, and so on. One context of work in administrative policy is "administrative RBAC", in which policy controls the permissions of roles, the membership of roles, and other elements of RBAC access-control state.

Here we study and extend the UARBAC model for administrative RBAC from the perspective of usability and expressiveness. Using tools from logic and program verification, we formulate UARBAC logically and develop an algorithm that produces "administrative plans" that achieve specified permissions through permitted actions. This work is closely related to work on the safety problem in administrative access control, but is intended to aid legitimate users in understanding how to achieve a desired access-control state. We then show how this machinery can be used so that administrative actions at any desired depth, and so plans as well, can be uniformly simulated in the existing UARBAC model.

1. INTRODUCTION

An access-control policy describes whether entities have permissions on actions in some application domain. An obvious question is: who has permission to change permissions? Without such "second-order" permissions, users could simply modify policy to obtain whatever permissions they desire. For example, suppose the dean of an academic department wants access to a potential student's application, but does not have permission. Without access control on the policy, the dean could simply modify permissions on the application, or could give herself the role of "member of the admissions committee", because users in this role have access to students' applications.

If second-order permissions are supported, then it makes sense to control them using third-order permissions, and so on. A policy that contains its own higher-order permissions is sometimes called an "administrative" access-control policy.

SACMAT'11, June 15–17, 2011, Innsbruck, Austria.

We briefly mention some desirable features of administrative access-control policy. First is *expressiveness*: it should be possible to express permissions of any order. For example, it should be possible to say that the dean of a department has the right to remove from a faculty member the right to give to a student permission to purchase a book using department funds. Second is *uniformity*: a single style of specifying permissions and a single method of enforcing them should be used. For example, in administrative RBAC, a single rule for defining permissions should be used at all levels of permissions. Third is *scope*: all aspects of policy should be controlled by policy. For example, permissions on modifying role membership, and the objects governed by policy, should be expressed in and controlled through policy. Finally is the *use of logic*: access-control states and permissions should be defined logically to allow existing theory and tools to be leveraged in policy design and analysis.

Work on administrative RBAC seeks to use RBAC-style mechanisms to define higher-order permissions on RBAC policies. This work has focused on two main questions. First: how should RBAC be applied to RBAC? (See, e.g., [22, 8, 18, 15].) For example, should administrative roles be separate from ordinary roles? Second: how difficult is it to analyze administrative RBAC policies? (See, e.g., [11, 26, 16, 13, 28].) For example, is it decidable (and if so at what cost) to determine whether a user can legally modify the policy to reach a specified state?

The question of whether one can make such modifications to a policy is sometimes called the "safety problem" [11]. The idea is that one would like to know whether certain policy states, seen as undesirable, can be reached by an "attacker".

Here we examine administrative RBAC in the context of the UARBAC [15] formalism. The goal of UARBAC is to improve the uniformity of administrative RBAC. In this paper we attempt to build on UARBAC in several ways:

We describe UARBAC using logic, providing improved uniformity across administrative and non-administrative actions in Section 2. We assess UARBAC from the point of view of the features we have listed above in Section 3. In Section 3, we then develop an algorithm to compute "administrative programs" for UARBAC that achieve specified permissions through permitted actions. And in Section 5, we show how permissions on administrative actions of a higher degree than supported in UARBAC can be simulated within UARBAC. We also discuss issues and perspectives for future work of our approach in Section 6. Our conclusions are in Section 7.

Our work on computing administrative programs is simi-

lar to work on the safety problem in administrative RBAC, but from a very different perspective. While the safety problem is concerned with preventing "attacks", we are concerned with helping legitimate users understand how they can achieve permissions through legal and expected administrative actions. For example, we would like a system to help the dean to learn that she can modify the policy state – in this case, membership of the admission committee – to access the application. In other words, our interest is not in reachability *per se*, but in the synthesis of plans that can help users achieve permissions they desire.

Related work

Much work on administrative RBAC inherits from the work on ARBAC97 [22]. Like most work on administrative RBAC, the aim in ARBAC97 is to administer RBAC with RBAC. However, the goal is not fully achieved, as a new class of administrative roles is introduced, and new relations for administrative permissions are defined. Furthermore, only two levels of administration are supported. ARBAC99 [23] and ARBAC02 [18] are refinements of ARBAC97 that attempt to allow for more expressive policy, but go no further than ARBAC97 in the aim of using RBAC to administer RBAC. Work on SARBAC [8] and A-ERBAC [12] are other variants, both using concepts of groups, or "scopes", to provide expressiveness beyond ARBAC97. A-ERBAC eliminates any technical distinction between normal and administrative roles. UARBAC [15] also contains no new class of role, and eliminates new relations for administrative permissions, instead using derived permissions for administrative actions. However, a new category of actions are introduced that are neither application-specific nor modify the access-control state. In the work on extended privilege inheritance in RBAC [9], administrative actions are defined simply as a broader class of actions under control of RBAC, and administrative privileges are inherited via roles just as in RBAC. None of these last-mentioned works support administrative policy analysis.

Trust management [6, 14], which focuses on delegation, is like administrative RBAC in that delegations are actions that have associated permissions, and that modify access-control state. In most trust-management frameworks, permissions on delegation are only loosely controlled through policy, but are controlled more through the "discretion" of delegators. From the point of view of our work, trust management is quite limited as a form of administrative access control. More precise controls could be put on delegation powers; for example in controlling who can delegate, who can grant delegation powers, etc.

There is much work on the analysis of the "safety problem" in access control. We shall not discuss the body of work concerning analysis of fixed (i.e., non-administrative) policies. However, we note that the use of model checking in the analysis of access-control policies (e.g., [13]) is useful because there is a wide range of safety questions that can be asked of a policy.

Other work looks at the safety problem in administrative access control. In [11] a simple, generic model of administrative access control is defined, and the safety problem for this model is shown to be undecideable. In this model there is no bound on the size of the access-control state.

The relevance and correctness of the work in [11] is examined in [16, 17]. Most importantly, in [16] it is argued that the model of [11] is incomparable to Discretionary Access Control (DAC) models, and the decidability results of [11] are therefore not always applicable to DAC. A cubic-time algorithm is presented for deciding safety in the Graham-Denning DAC scheme.

In [10, 30] the authors present a logic-based formalism for administrative access control, a means of expressing goals, and an algorithm for checking the reachability of goals. Termination of the algorithm is guaranteed by finiteness of the access-control state.

In [28], algorithms and complexity results are provided for safety analysis in ARBAC97 policies. In [27], the authors defined a parameterized form of ARBAC, named PARBAC, and show the same kinds of results for policies of this model. The algorithms rely on the principle of *separate administration*, which partitions roles into administrative and non-administrative ones.

In [3, 4], authentication logics are developed in which permissions are derived from inference rules that separate updates of access-control states from constraints that decide the applicability of such rules. This separation is hoped to aid comprehension and maintainability of authorization policies. In [3], a proof system is the basis for a goal-oriented search algorithm that finds minimal sequences of transactions that lead to desired authorization state.

In [24], the authors study the analysis of policies written in ARBAC. They identify a connection between the analysis of access-control policies and planning as developed in artificial intelligence. They then exploit this connection to prove complexity results for the reachability problem in ARBAC. In particular, they show that plans for reaching specified target states in ARBAC may not have polynomial size.

The work in [31, 21] defines and studies a modeling language X-*Policy*, designed for applications within the space of collaborative web-based systems. Permissions focus on who can read or write what resources under which constraints. Model-checking techniques are being proposed for automated analysis of security properties.

2. OVERVIEW OF UARBAC

UARBAC is described in [15], referred to as "Li/Mao" henceforth. The key idea of UARBAC is that administration of RBAC can be done using RBAC itself: "permissions about users and roles are administered in the same way as permissions about other kinds of objects" [15]. This idea is clearly related to our requirement of using a single specification system and a single enforcement method.

Our technical presentation of UARBAC varies from [15], but retains the essential technical features. We assume the existence of a finite set C of class names, and a finite set Obj of objects. The set of class names includes the names role and user but may contain domain-specific names as well. For example, book might be a class name, and "Manager" (an object of class role), "Larry" (an object of class user), and "Firewalls" (an object of class book) might be objects.

We also assume a function $classOf : Obj \to C$ giving the class of each object. The function $Dom : C \to 2^{Obj}$ is derived from $classOf$ as follows

$$Dom(c) = \{x \in Obj \mid classOf(x) = c\}$$

When we refer to a *role*, we mean an object in $Dom(\texttt{role})$, and similarly for *user*. Set $Dom(\texttt{role})$ is assumed to contain

$$
\begin{array}{rcl}
\alpha & ::= & \beta \mid \gamma \\
\beta & ::= & f(c) \mid f(x,c) \\
\gamma & ::= & op(\texttt{PA},(\alpha,r)) \mid op(\texttt{RH},(r_1,r_2)) \mid \\
 & & op(\texttt{OB},(c,x)) \mid op(\texttt{UA},(u,r))
\end{array}
$$

Figure 1: Abstract syntax of actions α. An action α can be a basic action β or an administrative action γ. Symbol f ranges over operator names, op ranges over administrative operators $\{add, remove\}$, x ranges over object names, c ranges over class names, u ranges over elements of $Dom(\texttt{user})$, and r ranges over elements of $Dom(\texttt{role})$.

the special administrative role name sso, which stands for "system security officer".

Li/Mao define a set of actions that can be requested. The abstract syntax of actions is defined in Figure 1. In the figure α is an action, γ is an administrative action, and β is a basic (i.e. non-administrative) action. The names in $\{\texttt{OB}, \texttt{UA}, \texttt{PA}, \texttt{RH}\}$ pertain to the functions and relations defined by an access-control state (see below). An operator name can be either an application-specific operation, such as read for a file read, or the name of one of UARBAC's built-in operations: admin, empower, grant, and create.

The syntax of Fig. 1 supports administration of actions of any "order". An example of an administrative action is $\mathsf{add}(\texttt{PA}, (\mathsf{buy}(book), \mathsf{Engineer}))$. This action extends the role-permission relation PA so that action $\mathsf{buy}(book)$ is permitted for role Engineer.

Li/Mao support only administrative actions that do not contain administrative actions. For example, action

$$add(\texttt{PA}, (buy(book), \mathsf{Engineer}))$$

is supported in Li/Mao, but the "second-order" administrative action

$$add(\texttt{PA}, (add(\texttt{PA}, (buy(book), \mathsf{Engineer})), \mathsf{Manager}))$$

is not. In other words, the syntax of actions supported by Li/Mao is like that of Fig. 1 except that clause $op(\texttt{PA}, (\alpha, r))$ of γ is replaced by $op(\texttt{PA}, (\beta, r))$. Below, we often write a to denote *atomic* administrative actions, which are those that do not contain administrative actions themselves.

An *access-control state* (or *AC state* for short) is a set of atoms, each of one of the following forms:

$$atom ::= OB(c,x) \mid UA(u,r) \mid PA(\beta,r) \mid RH(r_1,r_2)$$

where c ranges over class name, x over object names, u over users, r over roles, and β over non-administrative actions. We write s_{OB} for the set of OB atoms in s, and similarly for the other atoms.

Intuitively, the elements of an AC state define some relations and functions. The OB atoms define the set $\{x \mid OB(c,x)\}$ of objects that "currently exist" for class c. The UA atoms define a user/role relation $\{(u,r) \mid UA(u,r) \in s\}$. The PA atoms define the set $\{\beta \mid PA(\beta,r)\}$ of basic actions permitted for role r. The RH atoms define a role hierarchy relation, which is required to be irreflexive and acyclic. We write $\mathcal{AC}$ for the set of all AC states and write $\mathcal{R}$ as a shorthand for $Dom(\texttt{role})$.

```
OB(role, Director)   // other roles elided
OB(user, Larry)      // other users elided
OB(book, b)
RH(Director, Manager)
RH(Manager, Engineer)
UA(Larry, Director)
UA(Lisa, Manager)
UA(Greg, Engineer)
PA(admin(book),Manager)
PA(empower(role,Engineer), Manager)
PA(admin(role,Manager), Director)
```

Figure 2: An example of an AC State.

Example. Figure 2 depicts a simple AC state. The first OB atoms indicate the existence of roles, users, and other objects. The RH atoms define that Directors lie above Managers, and Managers above Engineers in the role hierarchy. The UA atoms show that Larry is a Director, Lisa is a Manager, and Greg is an Engineer. Finally, the PA atoms give permissions on basic actions. For example, a Manager can administer book objects. **(End of Example.)**

We write $perm(\alpha, r)$ to indicate whether action α is permitted by a member of role r. For basic actions, it is derived as follows:

$$
\begin{array}{rcll}
perm(f(c),r) & = & PA(f(c),r) & \\
perm(f(c,o),r) & = & PA(f(c,o),r) \vee PA(f(c),r) & (1)
\end{array}
$$

Figure 3 defines $perm$ for atomic administrative actions. The figure does not include conditions related to basic well-formedness of actions, such as that the role parameter of an action is a valid role.

Example. Returning to the example of Fig. 2, we consider some actions of users governed by the policy. First, suppose that Lisa wishes to give Engineers the permission to purchase books. To do so, Lisa would attempt to perform the administrative action

$$\mathsf{add}(\texttt{PA}, (\mathsf{buy}(book), \mathsf{Engineer}))$$

By Figure 3, Lisa, as Manager, has permission to perform this operation, because

$$\texttt{PA}(\mathsf{empower}(\texttt{role}, \mathsf{Engineer}), \mathsf{Manager})$$
$$\texttt{PA}(\mathsf{admin}(book), \mathsf{Manager})$$

both hold. As a result, the atom $\texttt{PA}(\mathsf{buy}(book), \mathsf{Engineer})$ is added to the AC state. Now Greg has permission to buy book b, using basic action

$$\mathsf{buy}(book, b)$$

The AC state is unaffected by this action. Continuing with our example, suppose Larry is unhappy that Engineers now have the right to buy books. Larry can perform the action

$$\mathsf{remove}(PA, (admin(book), \mathsf{Manager}))$$

that removes the power of Managers to administer books because, by Fig. 3, to do so it is enough that atom $\texttt{PA}(\mathsf{empower}(role, Manager), \mathsf{Director})$ belongs to the AC state. The effect of this action is to remove $\texttt{PA}(\mathsf{admin}(book), \mathsf{Manager})$ from the AC state. However, it does not remove the permission that Engineers have to buy books. It may seem that,

a	$perm(a, r)$
$add(\texttt{OB}, (c, o))$	$perm(\mathsf{create}(c), r)$
$remove(\texttt{OB}, (c, o))$	$perm(\mathsf{admin}(c, o), r)$
$add(\texttt{UA}, (u, r'))$	$perm(\mathsf{grant}(\texttt{role}, r'), r) \wedge perm(\mathsf{empower}(user, u), r)$
$remove(\texttt{UA}, (u, r'))$	$perm(\mathsf{admin}(\texttt{role}, r'), r) \vee perm(\mathsf{admin}(user, u), r) \vee$ $(perm(\mathsf{grant}(\texttt{role}, r'), r) \wedge perm(\mathsf{empower}(user, u), r))$
$add(\texttt{RH}, (r_1, r_2))$	$perm(\mathsf{grant}(\texttt{role}, r_1), r) \wedge perm(\mathsf{empower}(\texttt{role}, r_2), r)$
$remove(\texttt{RH}, (r_1, r_2))$	$perm(\mathsf{admin}(\texttt{role}, r_1), r) \vee perm(\mathsf{admin}(\texttt{role}, r_2), r) \vee$ $(perm(\mathsf{grant}(\texttt{role}, r_1), r) \wedge perm(\mathsf{empower}(\texttt{role}, r_2), r))$
$add(\texttt{PA}, (f(c, o), r'))$	$perm(\mathsf{admin}(c, o), r) \wedge perm(\mathsf{empower}(\texttt{role}, r'), r)$
$remove(\texttt{PA}, (f(c, o), r'))$	$perm(\mathsf{admin}(c, o), r) \vee perm(\mathsf{admin}(\texttt{role}, r'), r)$
$add(\texttt{PA}, (f(c), r'))$	$r = \texttt{sso}$
$remove(\texttt{PA}, (f(c), r'))$	$r = \texttt{sso} \vee perm(\mathsf{admin}(\texttt{role}, r'), r)$

Figure 3: Atomic administrative action a of UARBAC (left column) and condition $perm(a, r)$ (right column) specifying whether role r has permission to perform a in UARBAC. These conditions make use of permissions on *pseudo-actions* admin, create, empower, and grant.

as a member of a role above Manager, a Director should be able to remove administrative powers over books because Managers can do so. But UARBAC does not fix whether administrative actions are inherited according to the role hierarchy. **(End of Example.)**

Our formalization of UARBAC differs from the presentation in [15]. We use the single concept of action for both administrative and non-administrative operations; Li/Mao use the term "permission" for our basic action, and the term "administrative operation" for our atomic administrative action. We define access-control states simply as models of first-order logic, which is helpful in defining logics of permissions. In some cases we define the semantics of administrative actions differently than in [15]. For example, Li/Mao define the administrative operation createObject such that the user performing the operation gets administrative permissions on the object. For simplicity we did not model this aspect, but we could do so, and the change would not impact the methods developed later in this paper.

3. ASSESSING UARBAC

We now list some of the features and limitations of UARBAC (in the version of [15]) most relevant to our interest in administrative policy. First, in UARBAC there are only two levels of action: basic actions, and atomic administrative actions. As explained already, one cannot express nested administrative actions in UARBAC.

UARBAC has two types of basic actions: application-specific ones and the "pseudo-actions" create, admin, grant, and empower. These actions are curious because they are not application-specific; but they are not administrative either, as they do not modify the AC state. Indeed, they are not really actions (hence our use of "pseudo"), as they can neither be requested nor performed. Instead, they serve only to define permissions on administrative actions.

A closely-related point is that permissions in UARBAC cannot be set directly on administrative actions. Instead, permissions on administrative actions are derived in a fixed manner from permissions on pseudo-actions. It is helpful to think of pseudo-actions as a mechanism to define constraints between administrative permissions. For example, because of the way administrative permissions are derived from permissions on pseudo-actions, in UARBAC any role with permission to remove an object also has permission to give some other role permission on an action on that object. This is reflected in the second disjunct in (1). Also, permission conditions on actions built from operators f are uniform in, and independent from, the choice of f.

One of our desiderata for administrative policy is that a single method be used for determining permissions on administrative and non-administrative permissions. In Li/Mao it is stated that UARBAC allows RBAC mechanisms to administer RBAC. But this is true only in a limited way, since permissions on administrative actions and permissions on basic actions are not defined by a single rule, as one might expect in an RBAC framework.

Permissions on basic actions follow the usual RBAC approach: a role r has permission on an action β if (r', β) belongs to the current permission relation, where r' is r itself or a lesser role. But according to Li/Mao, there is no fixed notion of permissions on administrative actions. Also, since permissions cannot be set for administrative actions, the rule just described for basic actions cannot be applied.

Finally, Li/Mao do not describe how one might analyze UARBAC policies.

4. PLANS AS AUTHORIZED PROGRAMS

In administrative access control, a common scenario is that a user modifies the policy – according to permissions in the policy itself – to add or remove permissions of other users. However, a user can also attempt to modify the policy to increase her own or someone else's permissions. Thus, a user can ask: if I do not have a permission now, do I have permission on modification actions to the policy that will give me my desired permission? This problem is a variant of the widely-studied reachability problem in access control. Here we consider a version of the problem in the context of UARBAC. (Related work on the reachability problem in access control was discussed in Section 1.)

In the remainder of this section we present a logic of permissions, allowing one to express complex permissions in terms of basic permissions. Then we define a "program" as a sequence of atomic administrative actions, and show how to synthesize programs from the current access-control state and a logical formula expressing the desired permissions. If any such program exists, all of the steps of the program

$$\begin{array}{rcl} \phi & ::= & A(t) \mid t_1 = t_2 \mid \neg\phi \mid \\ & & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ t & ::= & x \mid f(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \end{array}$$

$$\begin{array}{rcl} s \models A(t) & \stackrel{\text{def}}{=} & A([\![t]\!]) \in s \\ s \models (t_1 = t_2) & \stackrel{\text{def}}{=} & [\![t_1]\!] = [\![t_2]\!] \end{array}$$

Figure 4: At left is the abstract syntax of formulas ϕ and terms t, where x ranges over constant symbols, f ranges over function symbols, and A ranges over predicate symbols. The meaning of terms is just their syntax. At right is semantics of formulas; $s \models \phi$ means that access-control state s satisfies formula ϕ.

will be permitted, and the program will terminate with the desired permissions in place.

4.1 Logical Formulas

We define a simple quantifier-free and variable-free first-order logic on permissions. Informally, a formula in the logic is built up from atoms of form $A(t_1, \ldots, t_m)$ and $t_1 = t_2$ using propositional connectives. An atom $A(t_1, \ldots, t_m)$ holds of an AC state s simply if $A(t_1, \ldots, t_m)$ is in s. An atom $t_1 = t_2$ holds if t_1 and t_2 denote syntactically identical terms.

For simplicity we define the logic generically, and then instantiate for UARBAC. The vocabulary of the logic consists of a set of constant symbols, a set of function symbols, and a set of predicate symbols. The abstract syntax of terms and formulas of the logic is then defined in Fig. 4 (left). We also allow symbol `true` as a formula; it can be regarded as shorthand for $t = t$, where t is any term.

Formulas of the logic are interpreted relative to models that consist of a set of atoms of the form $A(t)$. We use a Herbrand-style semantics by letting the meaning $[\![t]\!]$ of terms t be just their syntax t:

$$[\![t]\!] \stackrel{\text{def}}{=} t \tag{2}$$

The semantics of formulas is defined in Fig. 4 (right). We write $s \models \phi$ if a model s satisfies a formula ϕ. We have omitted the clauses for the propositional connectives and truth constants, which are standard.

To instantiate this logic for UARBAC, we let the constant symbols be class names and object names, and let the predicate symbols be the elements of $\{OB, UA, PA, RH\}$. An access-control state then becomes a logical model.

An example formula of the logic is

$$\mathtt{UA}(Greg, \mathsf{Engineer}) \wedge \mathtt{PA}(\mathsf{buy}(book), \mathsf{Engineer})$$

which holds of an AC state if both

$$\mathtt{UA}(Greg, Engineer)$$
$$\mathtt{PA}(buy(book), Engineer)$$

exist in that state. Intuitively, Greg has permission to buy books, since user *Greg* is an Engineer. Another example is

$$\mathtt{PA}(\mathsf{admin}(\mathsf{File}, f_1), \mathsf{Doctor}) \wedge \mathtt{PA}(\mathsf{empower}(\mathsf{Nurse}, \mathsf{Doctor})$$

which, by Fig. 3, expresses that a doctor can modify the AC state so that a nurse can perform an action on file f_1. Note that all conditions on the right of Fig. 3 are well-formed formulas of this logic.

A *positive* formula is one containing no negation symbols. A *negation-flat* formula is one in which negation does not appear within the scope of conjunction, as defined in Fig. 5.

$$\begin{array}{rcl} \phi & ::= & \delta \mid \neg A(t) \mid \phi_1 \vee \phi_2 \mid \psi \\ \psi & ::= & \delta \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \\ \delta & ::= & A(t) \mid t_1 = t_2 \mid t_1 \neq t_2 \mid \end{array}$$

Figure 5: Syntax of negation-flat formulas ϕ, where the syntax of terms is as in Fig. 4

$$p ::= () \mid \gamma \mid p_1; p_2$$

$$\begin{array}{rcl} [()](s) & \stackrel{\text{def}}{=} & s \\ [add(A,t)](s) & \stackrel{\text{def}}{=} & s \cup \{A(t)\} \\ [remove(A,t)](s) & \stackrel{\text{def}}{=} & s \setminus \{A(t)\} \\ [p_1; p_2](s) & \stackrel{\text{def}}{=} & [p_2]([p_1](s)) \end{array}$$

Figure 6: Syntax of programs (top), where γ ranges over administrative actions. We write $\mathcal{P}$ for the set of all programs. Semantics of programs (bottom) as total function of type $\mathcal{P} \to (\mathcal{AC} \to \mathcal{AC})$, mapping programs to transformers of AC states (bottom).

4.2 Programs

A *program* is a sequence of atomic administrative actions. The syntax and semantics of programs is shown in Fig. 6. For the *add* and *remove* operators, each element A of the AC state is understood as a set; if A is a function or relation, we treat it as the graph of that function or relation. For example, $add(\mathtt{PA}, (\alpha, r))$ interprets PA as a binary relation and means to add pair (α, r) to it.

A program is *positive* if it contains no *remove* actions. Positive programs preserve positive formulas.

PROPOSITION 1. *For all positive programs p, positive formulas ϕ, and AC states s: if $s \models \phi$ then $p(s) \models \phi$.*

We now define a permission condition on programs. The idea is that a program is permitted if its first action is, and if performing the action leads to an AC state where the second action is permitted, etc.

We formalize the notion of program permissions using function wp, defined in Fig. 7. Formula $wp(p, \phi)$ is the *necessary and sufficient* condition required of an AC state so that ϕ holds after program p executes from that state. For example, for

$$\begin{array}{rcl} \phi & \stackrel{\text{def}}{=} & \mathtt{PA}(\mathsf{buy}(b2, book),\ \mathsf{Engineer}) \\ p & \stackrel{\text{def}}{=} & \mathsf{add}(\mathtt{PA}, (\mathsf{buy}(b1, book),\ \mathsf{Engineer})) \end{array}$$

$$
\begin{array}{rcl}
wp((), \phi) & \stackrel{\text{def}}{=} & \phi \\
wp(add(A,t), A(t')) & \stackrel{\text{def}}{=} & A(t') \vee (t = t') \\
wp(add(A,t), A'(t')) & \stackrel{\text{def}}{=} & A'(t') \qquad (A \neq A') \\
wp(add(A,t), t_1 = t_2) & \stackrel{\text{def}}{=} & t_1 = t_2 \\
wp(add(A,t), \phi_1 \wedge \phi_2) & \stackrel{\text{def}}{=} & wp(add(A,t), \phi_1) \\
 & & \wedge\, wp(add(A,t), \phi_2) \\
wp(remove(A,t), A(t')) & \stackrel{\text{def}}{=} & A(t') \wedge (t \neq t') \\
wp(remove(A,t), A'(t')) & \stackrel{\text{def}}{=} & A'(t') \qquad (A \neq A') \\
wp(remove(A,t), t_1 = t_2) & \stackrel{\text{def}}{=} & t_1 = t_2 \\
wp(remove(A,t), \phi_1 \wedge \phi_2) & \stackrel{\text{def}}{=} & wp(remove(A,t), \phi_1) \\
 & & \wedge\, wp(remove(A,t), \phi_2) \\
wp(p_1; p_2, \phi) & \stackrel{\text{def}}{=} & wp(p_1, wp(p_2, \phi)) \\
wp(p, \neg\phi) & \stackrel{\text{def}}{=} & \neg wp(p, \phi)
\end{array}
$$

$$
\begin{array}{rcl}
perm((), r) & \stackrel{\text{def}}{=} & \texttt{true} \\
perm(\gamma, r) & & \text{Defined in Fig. 3} \\
perm(p_1; p_2, r) & \stackrel{\text{def}}{=} & perm(p_1, r) \wedge wp(p_1, perm(p_2, r))
\end{array}
$$

Figure 7: Definition of functions $wp\colon \mathcal{P} \times \mathcal{L} \to \mathcal{L}$ and $perm\colon \mathcal{P} \times \mathcal{R} \to \mathcal{L}$.

the condition $wp(p, \phi)$ needed to establish that ϕ holds after program p executes is

$$\mathtt{PA}(\mathsf{buy}(b2, book),\ \mathsf{Engineer}))\ \vee\ b1 = b2.$$

We now state the correctness of wp formally.

THEOREM 1. *Let s be an AC state, p be a program, and ϕ be a formula. Then $s \models wp(p, \phi)$ iff $p(s) \models \phi$.*

Using wp we can define $perm(p, r)$, which is the formula that holds of an AC state s if role r has permission to run program p. Function $perm$ is defined at the bottom of Fig. 7. The following is a sanity check on the definition of $perm$.

PROPOSITION 2. *Let p_1, p_2 be programs, r be a role, and s be an AC state. Then $s \models perm(p_1; p_2, r)$ iff both $s \models perm(p_1, r)$ and $p_1(s) \models perm(p_2, r)$.*

We define formula $\overline{wp}(p, r, \phi)$ to be

$$wp(p, \phi) \wedge perm(p, r).$$

Formula $\overline{wp}(p, r, \phi)$ holds at an AC state if role r has permission to run the program, and doing so leads to a state satisfying ϕ. We shall later use some simple properties of $\overline{wp}$, which we now list. For formulas ϕ_1 and ϕ_2, expression $\phi_1 \Rightarrow \phi_2$ holds if, for all AC states s, relation $s \models \phi_1$ implies $s \models \phi_2$. Also, $\phi_1 \Leftrightarrow \phi_2$ holds if $\phi_1 \Rightarrow \phi_2$ and $\phi_2 \Rightarrow \phi_1$ hold.

PROPOSITION 3. *Let p be a program, r be a role, and ϕ, ϕ_1, ϕ_2 be formulas. Then the following all hold:*

1. $\overline{wp}(p_1; p_2, r, \phi) \Leftrightarrow \overline{wp}(p_1, r, \overline{wp}(p_2, r, \phi))$
2. $\overline{wp}(p, r, \phi_1 \wedge \phi_2) \Leftrightarrow \overline{wp}(p, r, \phi_1) \wedge \overline{wp}(p, r, \phi_2)$
3. *If $(\phi_1 \Rightarrow \phi_2)$ then $(\overline{wp}(p, \phi_1) \Rightarrow \overline{wp}(p, \phi_2))$*
4. $\overline{wp}(add(A,t), r, A(t)) \Leftrightarrow perm(add(A,t), r)$
5. $\overline{wp}(remove(A,t), \neg A(t)) \Leftrightarrow perm(remove(A,t), r)$

4.3 Program Synthesis

The predicate $perm(\alpha, r)$ holds of an AC state if a user in role r has permission on action α. But in administrative access control, a user may wish to know not only if an action α can be performed – permission in the *strong* sense – but whether administrative actions could be performed that would change the AC state such that α could be performed. This is permission in the *weak* sense.

Fig. 8 defines function $progs$. The value $progs(s, r, \phi, \Phi)$ is a set of programs that r is permitted to run in s, and that all terminate in a state satisfying ϕ. Parameter Φ is used to ensure termination. The function can be understood as a backward-chaining planning algorithm, in which AC state s is the initial state, formula ϕ is the goal condition, and the possible actions are those listed in the left column of Fig. 3. Also, for any $P, Q \subseteq \mathcal{P}$ we write $P; Q$ for the subset $\{p; q \mid p \in P, q \in Q\}$ of $\mathcal{P}$ and write $P; q$ for $P; \{q\}$.

The algorithm works by considering and simplifying the structure of the goal condition ϕ. Suppose ϕ is a basic predicate of the form $A(t)$. Then the empty program () achieves the goal if ϕ is satisfied in initial state s; otherwise the goal can be achieved by action $add(A, t)$. If the latter, then the permission for action $add(A, t)$ is made the new goal. If ϕ is a formula of the form $\neg A(t)$, then the processing is similar, except $remove(A, t)$ is the action and the permission for $remove(A, t)$ is the new goal.

If the goal condition is a disjunction, then the problem is decomposed, with each disjunct becoming a goal, and similarly for conjunction. In the case of disjunction, a program for any disjunct is a program for the disjunction as a whole. For the case of conjunction, a program built sequentially from the programs for all conjuncts is a program for the conjunction as a whole. Intuitively, this treatment of conjunction works because the conjuncts must be positive, and therefore do not "interfere" with each other.

The next theorem establishes that the function $progs$ terminates, is sound, and is complete, in the sense that a program will be produced, if one exists, that will achieve the desired condition.

THEOREM 2. *Let p be a program, r be a role, s be an AC state, ϕ be a negation-flat formula, and Φ be a set of formulas. Then:*

1. *$progs(s, r, \phi, \Phi)$ terminates.*
2. *If $p \in progs(s, r, \phi, \Phi)$ then $s \models \overline{wp}(p, r, \phi)$.*
3. *If $progs(s, r, \phi, \emptyset) = \emptyset$, then there is no program $p \in \mathcal{P}$ such that $s \models \overline{wp}(p, r, \phi)$.*

One may think of the computed set $progs(s, r, \phi, \emptyset)$ as a tree with internal nodes "$\cup$" and ";", and leaves "$\emptyset$" and "$\{()\}$". Also, only branch points for $\wedge$ and $\vee$ compute subtrees that are *both* non-trivial. The complexity of $progs$ is therefore directly related to the width and depth of that tree. The depth is bounded by the well-founded ordering used in the termination analysis of this algorithm. The width

ϕ	$progs(s, r, \phi, \Phi)$
$A(t)$	$\begin{cases} \emptyset & \text{if } \phi \in \Phi \\ \{()\} & \text{if } \phi \notin \Phi \text{ and } s \models \phi \\ progs(s, r, perm(add(A,t), r), \Phi \cup \{\phi\}); add(A,t) & \text{otherwise} \end{cases}$
$\neg A(t)$	$\begin{cases} \emptyset & \text{if } \phi \in \Phi \\ \{()\} & \text{if } \phi \notin \Phi \text{ and } s \models \phi \\ progs(s, r, perm(remove(A,t), r), \Phi \cup \{\phi\}); remove(A,t) & \text{otherwise} \end{cases}$
$t_1 = t_2$	$\begin{cases} \{()\} & \text{if } s \models \phi \\ \emptyset & \text{otherwise} \end{cases}$
$t_1 \neq t_2$	$\begin{cases} \{()\} & \text{if } s \models \phi \\ \emptyset & \text{otherwise} \end{cases}$
$\phi_1 \vee \phi_2$	$progs(s, r, \phi_1, \Phi) \cup progs(s, r, \phi_2, \Phi)$
$\phi_1 \wedge \phi_2$	$progs(s, r, \phi_1, \Phi); progs(s, r, \phi_2, \Phi)$

Figure 8: Definition of function $progs$: $\mathcal{AC} \times \mathcal{R} \times \mathcal{L} \times 2^{\mathcal{L}} \to 2^{\mathcal{P}}$. Formula ϕ (left) ranges over elements on $\mathcal{L}$ in negation-flat form. The corresponding expression on the right defines $progs(s, r, \phi, \Phi)$ for $\Phi \subseteq \mathcal{L}$.

is bounded by the number of nestings of disjunctions and conjunctions encountered in the computation.

The correctness of function *progs* depends on parameter ϕ being in negation-flat form. Alternatively, one could write a brute-force synthesis algorithm that would cope with any formula in our logic. Also, function *progs* could easily be modified to return only a single program to achieve the specified goal, rather than what could potentially be a large set.

In what follows we often write $progs(s, r, \phi)$ as shorthand for $progs(s, r, \phi, \emptyset)$. Also, we write *Act* and *BAct* for the sets of all α, respectively β, generated in Fig. 1.

COROLLARY 1. *Let s be in $\mathcal{AC}$, r in $\mathcal{R}$, and α in Act. Then for any p in $progs(s, r, perm(\alpha, r))$ we have that $s \models perm(p; \alpha, r)$.*

Thus a user who wants to reach a specified state of permissions can use function *progs* to find a program whose execution will reach that state.

Example. We now illustrate the use of *progs* using the example of Figure 2. In Section 2 we explained that director Larry might not want managers to have permission to give engineer's permission to buy books. Consider how function *progs* could be used by Larry to find a program to reach this goal. His goal is:

$$\neg perm(add(\mathtt{PA}, (buy(book), \mathsf{Engr})), \mathsf{Mgr})$$

which states that a manager does not have permission to give Engineers book-buying permission (we use abbreviated role names in this example to save space). By the definition of *perm* this is equivalently:

$$\neg(\mathtt{PA}(admin(book), \mathsf{Mgr}) \wedge \mathtt{PA}(empower(role, E), \mathsf{Mgr}))$$

which by DeMorgan's law is equivalently:

$$\neg\mathtt{PA}(admin(book), \mathsf{M}) \vee \neg\mathtt{PA}(empower(role, E), \mathsf{Mgr}))$$

Let us refer to this formula as $\phi_1 \vee \phi_2$.

Next we try to find programs to reach this goal function *progs*: The function application $progs(s, \mathsf{Dir}, \phi_1 \vee \phi_2, \emptyset)$ expands to

$$progs(s, \mathsf{Dir}, \phi_1, \emptyset) \cup progs(s, \mathsf{Dir}, \phi_2, \emptyset)$$

Evaluating the first sub-expression yields:

$$progs(s, \mathsf{Dir}, perm(remove(\mathtt{PA}, (admin(book), \mathsf{Mgr})), \mathsf{Dir}), \phi_1); remove(\mathtt{PA}, (admin(book), \mathsf{Mgr}))$$

Evaluating the *progs* call, we have:

$$\begin{aligned} & progs(s, \mathsf{Dir}, \\ & \quad perm(remove(\mathtt{PA}, (admin(book), \mathsf{Mgr})), \mathsf{Dir}), \phi_1) \\ = \; & progs(s, \mathsf{Dir}, \\ & \quad \mathtt{PA}(admin(book), \mathsf{Dir}) \vee \mathtt{PA}(admin(role, \mathsf{Mgr}), \mathsf{Dir}), \phi_1) \\ = \; & progs(s, \mathsf{Dir}, \mathtt{PA}(admin(book), \mathsf{Dir}), \{\phi_1\}) \cup \\ & progs(s, \mathsf{Dir}, \mathtt{PA}(admin(role, \mathsf{Mgr}), \mathsf{Dir}), \{\phi_1\}) \end{aligned}$$

Note that $\mathtt{PA}(admin(role, \mathsf{Mgr}), \mathsf{Dir})$ is an atom of the AC state. Therefore

$$progs(s, \mathsf{Dir}, \mathtt{PA}(admin(role, \mathsf{Mgr}), \mathsf{Dir}), \{\phi_1\}) = \{()\}$$

This means that no action is needed to reach goal

$$PA(admin(role, \mathsf{Mgr}), \mathsf{Dir})$$

Therefore the empty program () is also contained in

$$progs(s, \mathsf{Dir}, perm(remove(\mathtt{PA}, (admin(book), M)), \mathsf{Dir}), \{\phi_1\})$$

and therefore we have both set memberships

$$remove(\mathtt{PA}, (admin(book), \mathsf{Mgr})) \in progs(s, \mathsf{Dir}, \phi_1, \emptyset)$$
$$remove(\mathtt{PA}, (admin(book), \mathsf{Mgr})) \in progs(s, \mathsf{Dir}, \phi_1 \vee \phi_2, \emptyset)$$

In short, using *progs*, Larry has learned he has permission to perform program

$$remove(\mathtt{PA}, (admin(book), \mathsf{Mgr}))$$

and doing so will result in goal condition

$$\neg perm(add(\mathtt{PA}, (buy(book), E)), \mathsf{Mgr})$$

holding in the AC state. **(End of Example.)**

5. SIMULATING NON-ATOMIC ADMINISTRATION IN UARBAC

UARBAC, as defined by Li/Mao in [15], only supports *atomic* administrative actions. We now show that there is a way to check, within Li/Mao's UARBAC, permissions of non-atomic administrative actions. Furthermore, there is a way for users (when permitted) to operate on the AC state to achieve these permissions.

The idea we use is that a modified notion of permission will be used for non-atomic administrative actions. To see

the idea, suppose that a director wishes to add permission for managers to remove permission for engineers to purchase books. This permission can't be expressed in Li/Mao's UARBAC. But director Larry can achieve the desired outcome if he can modify the AC state such that a manager can modify the AC state such that engineers *cannot* modify the control state to get permission to purchase books. We can express and check such "weak" permissions.

What property should we want weak permissions to possess? We can make an analogy to a property that holds of permissions on atomic administrative actions: if an operation to add (β, r) to $\mathtt{PA}$ is permitted, then applying the action leads to a state in which β is permitted for r. For example, if AC state s satisfies $\mathtt{PA}(add(\mathtt{PA}, (\alpha, r)), r')$ then modified state $add(\mathtt{PA}, (\alpha, r))$ satisfies $\mathtt{PA}(\alpha, r)$. A similar property holds of *remove* actions.

Weak permissions should possess a similar property: if an operation to add (α, r) to $\mathtt{PA}$ is weakly permitted, then there should exist a permitted program p that leads to a state in which α is weakly permitted for r. For example, if s satisfies $perm'(add(\mathtt{PA}, (\alpha, r)), r')$, then there exists a program p such that s satisfies $perm(p, r')$ and $p(s)$ satisfies $perm'(\alpha, r)$. Here $perm'$ is a weak permission predicate, and α can be an administrative action.

A difficulty with this property is that it uses the idea of existential quantification over *programs*. We would like to capture that property logically, and so need a way to express in logic that there exists a program with which a member of a role can reach an AC state satisfying some condition.

The following theorem states that function Ep, defined in Fig. 9, when given a role r and formula ϕ, returns formula $Ep(r, \phi)$ that holds at an AC state s if there exists a program p such that $s \models \overline{wp}(p, r, \phi)$.

THEOREM 3. *Let s be an AC state, r be a role, and ϕ be a formula. Then $s \models Ep(r, \phi)$ iff there exists a program p such that $s \models \overline{wp}(p, r, \phi)$.*

Using Ep we define function $perm'$ at the bottom of Fig. 9. Logical formula $perm'(\alpha, r)$ expresses weak permission for role r on action α. This notion of weak permission satisfies the property that we have said we expect of it.

PROPOSITION 4. *Let s be an AC state. Then:*

1. *For any α of form $add(\mathtt{PA}, (\alpha', r'))$ with $s \models perm'(\alpha, r)$ there exists a program p such that $s \models perm(p, r)$ and $p(s) \models perm'(\alpha', r')$.*
2. *For any α of form $remove(\mathtt{PA}, (\alpha', r'))$ such that $s \models perm'(a, r)$ there exists a program p such that $s \models perm(p, r)$ and $p(s) \not\models perm'(\alpha', r')$.*

It should be highlighted that $Ep(r, \phi)$ is a formula of our logic, and so we have captured the notion of reachability through a policy as a formula that can be checked of an AC state. Ep is closely tied to function $progs$: for all s r, and ϕ it is the case that $s \models Ep(r, \phi)$ iff there is some p in $progs(s, r, \phi)$. Similarly, $perm'(\alpha, r)$ is a formula of our logic, and so a weak permission can be checked of an AC state.

To summarize, we have shown how permissions on non-atomic administrative actions, absent in Li/Mao's UARBAC, can be simulated in it. Next we define the effect of the application of a non-atomic administrative action in Li/Mao's UARBAC.

For an administrative action γ, we define in Fig. 10 the desired effect of γ as a postcondition $post(\gamma)$. The definition says that by performing a non-atomic *add* action one seeks to enable the given action, and by performing a non-atomic *remove* action one seeks to disable it.

Now, let γ be any administrative action. A user in role r can use any program in $progs(s, r, post(\gamma))$ to accomplish the desired effect of action γ.

COROLLARY 2. *Let s be an AC state, r and r' be roles, and a be an administrative action. Then:*

1. *For every p in $progs(s, r, post(add(r', a)))$ we have that $p(s) \models perm'(r', a)$.*
2. *For every p in $progs(s, r, post(remove(r', a)))$ we have $p(s) \not\models perm'(r', a)$.*

6. DISCUSSION

We now discuss some issues of our approach, and point out avenues for future work that these issues provide.

6.1 Plans for Roles Versus Plans for Users

Our technical development supports plans relative to roles. This answers questions such as "Can a Director perform administration so that Engineers can no longer buy books?"

It is obviously of interest to ask and answer similar questions at the level of users. For example, we may ask "Can Colin perform administration so that Engineers can no longer buy books?" and the variant "Can Colin perform administration so that Barbara can no longer buy books?"

Our technical development can be adjusted and extended in order to accommodate support for such user-centric plans. We hint at how this can be realized; a complete exposition is deferred to a full paper. The predicates $perm(\alpha, r)$, $\overline{wp}(p, r, \phi)$, and $progs(s, r, \phi, \Phi)$ change so that roles r are replaced with users u. For example, we would set

$$perm(p_1; p_2, u) \stackrel{\text{def}}{=} perm(p_1, u) \wedge wp(p_1, perm(p_2, u))$$

where it is crucial that programs p_1 and p_2 can be performed by the same user u but potentially in different roles.

Similarly, the definition of $perm(\alpha, u)$ reflects the RBAC model that permissions for users are mediated through permissions on their roles. Thus, we would set

$$perm(a, u) \stackrel{\text{def}}{=} \bigvee_{r \in \mathcal{R}} \mathtt{UA}(u, r) \wedge perm(a, r)$$

for atomic administrative actions a.

6.2 Atomicity of Plan Execution

An authorized synthesized program is only a solution to our planning problem if it can execute without interference. But the access-control architecture may be such that access requests that are not part of the plan are interleaved with the plan execution and potentially granted. This may in fact corrupt the aim of plan execution. There are at least two responses of interest to this concern.

One response would be to use techniques developed for run-time verification [2]. Whenever a plan wants to execute the next step and finds it unauthorized, one would re-synthesize a desired plan from the current AC state and (if such a plan exists) execute that new plan.

$$
\begin{aligned}
Ep(r, t_1 = t_2) &\stackrel{\text{def}}{=} t_1 = t_2 \\
Ep(r, A(t)) &\stackrel{\text{def}}{=} A(t) \vee Ep(r, perm(add(A, t), r)) \\
Ep(r, \neg A(t)) &\stackrel{\text{def}}{=} \neg A(t) \vee Ep(r, perm(remove(A, t), r)) \\
Ep(r, \phi_1 \wedge \phi_2) &\stackrel{\text{def}}{=} Ep(r, \phi_1) \wedge Ep(r, \phi_2) \\
Ep(r, \phi_1 \vee \phi_2) &\stackrel{\text{def}}{=} Ep(r, \phi_1) \vee Ep(r, \phi_2)
\end{aligned}
$$

$$
\begin{aligned}
perm'(\alpha, r) &\stackrel{\text{def}}{=} Ep(r, perm(\alpha, r)) \ (\alpha \text{ basic or atomic administrative}) \\
perm'(add(\mathtt{PA}, (\alpha, r')), r) &\stackrel{\text{def}}{=} Ep(r, perm'(\alpha, r'))) \\
perm'(remove(\mathtt{PA}, (\alpha, r')), r) &\stackrel{\text{def}}{=} \neg Ep(r, perm'(\alpha, r')))
\end{aligned}
$$

Figure 9: Definition of functions Ep and $perm'$.

$$
\begin{aligned}
post(add(A, x)) &= A(x) \qquad \text{(atomic administrative case)} \\
post(remove(A, x)) &= \neg A(x) \qquad \text{(atomic administrative case)} \\
post(add(\mathtt{PA}, (\alpha, r))) &= perm'(\alpha, r) \\
post(remove(\mathtt{PA}, (\alpha, r))) &= \neg perm'(\alpha, r)
\end{aligned}
$$

Figure 10: Definition of function $post$. Formula $post(\gamma)$ gives the desired effect of administrative action γ.

Another response would be to use techniques developed for design synthesis [19]. The idea is here to try to compute a plan that will lead to a desired goal state no matter what actions the environment may choose to do during plan execution. A plan would then not be a sequence of actions but a tree of such actions, technically, a winning strategy in an infinite 2-person game played between a system (the user who wants to get to a goal state) and an environment (all other or a set of "hostile" users).

6.3 Coalition Plans

Agent-based approaches to verification and validation, e.g. the temporal logic ATL [1], can compute reachability under the assumption that a designated coalition of agents collaborate in order to reach the target set of states.

In our context, agents are users. In a positive reading, coalitions would analyze whether they can collaborate in order to realize desired permissions to get necessary work done. In a negative reading, coalitions could do the same analysis to determine whether permissions could be realized with which insider attacks [20] could be launched. As most scientific methods, our approach is agnostic to these interpretations.

Adapting our approach to support coalitions is straightforward for plan synthesis. In the reachability problem we considered above, we would simply consider every action that is authorized and involves a role (or user) from the given coalition.

6.4 Use of Constraints

Constraints are often needed in the enforcement of access control. Seperation of duty, e.g., may require that two actions be performed by different users. RBAC models may support, for example, user/role constraints and role activation constraints. This raises the question of whether authorized plans can be synthesized that meet such constraints.

We think that this problem is related to that of computing authorized work-flow schemes [5]. Such schemas have tasks that need to be assigned to authorized users, where certain tasks must precede certain others and where constraints on user/task assignment have to be met. Computing solutions for such schemas is typically NP-complete or NP-hard [29].

We believe that, for an important class of constraints such as those in [7], the synthesis of authorized, constrained plans for higher-order permissions can be achieved by satisfiability checking of suitable fragments of linear-time propositional temporal logic, e.g. the NP-complete fragment of LTL [25].

Another question pertaining to constraints is whether our version of UARBAC can be extended so that permissions are enriched with pre-conditions over authorization state. The use of such constraints is also expected to be useful when trying to extend our work to the parameterized version of UARBAC in [15].

7. CONCLUSIONS

In this paper we studied the issue of whether and how standard tools from planning and program verification can provide needed support for users of access-control systems that are governed not just by basic permissions, but also by higher-order (so called "administrative") permissions.

The support we had in mind was to answer whether users can reach specific access-control goal states and, if so, how planning can synthesize administrative programs whose execution is authorized and will reach a desired goal state.

We focused our attention on the RBAC family of access-control models and identified the UARBAC model as a promising candidate for developing such user support. The principal technical contributions of this paper, therefore, were a logical reconstruction of UARBAC that not only al-

lows for such support, but also provides a uniform enforcement rule, and the ability to simulate higher-order administrative actions within the original UARBAC model.

Acknowledgements

Glenn Bruns and Kumar Avijit were both supported in part by US National Science Foundation grant 0244901. We thank the anonymous reviewers for their detailed and very thoughtful comments.

8. REFERENCES

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. of the 38th Ann. Symp. on Found. of Computer Science*, 1997.

[2] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 2009. in press.

[3] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. In *Proc. of ESORICS 2007*, pp. 203-218, LNCS 4734, Springer, 2007.

[4] M. Y. Becker. Specification and Analysis of Dynamic Authorisation Policies. In *Proc. of Comp. Security Found. Symp.*, pp. 203-217, IEEE, 2009.

[5] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.* 2:65–104, February 1999.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the 1996 IEEE Symposium on Security and Privacy*, pp. 164–173. IEEE Computer Society Press, 1996.

[7] J. Crampton. A reference monitor for workflow systems with constrained task execution. In *Proc. of SACMAT '05*, pp. 38–47. ACM, 2005.

[8] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Trans. Inf. Syst. Secur.* 6:201–231, May 2003.

[9] M. A. C. Dekker, J. G. Cederquist, J. Crampton, and S. Etalle. Extended privilege inheritance in RBAC. In *Proc. of ASIACCS*, pp. 383–385. ACM, 2007.

[10] D. P. Guelev, M. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Information Security*, LNCS 3225. Springer, 2004.

[11] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM* 19:461–471, August 1976.

[12] A. Kern, A. Schaad, and J. Moffett. An administration concept for the enterprise role-based access control model. In *Proc. of SACMAT '03*, pp. 3–11. ACM, 2003.

[13] E. Kleiner and T. Newcomb. On the decidability of the safety problem for access control policies. *Electron. Notes Theor. Comput. Sci.* 185:107–120, 2007.

[14] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.* 6:128–171, Feb. 2003.

[15] N. Li and Z. Mao. Administration in role-based access control. In *Proc. of ASIACCS*, pp. 127–138. ACM, 2007.

[16] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *SP '05: Proc. of the 2005 IEEE Symposium on Security and Privacy*, pp. 96–109, 2005. IEEE Computer Society.

[17] N. Li and M. V. Tripunitara. The foundational work of Harrisson-Ruzzo-Ullman revisited. Technical Report CERIAS TR 2006-33, Purdue University, 2006.

[18] S. Oh, R. Sandhu, and X. Zhang. An effective role administration model using organization structure. *ACM Trans. Inf. Syst. Secur.* 9(2):113–137, 2006.

[19] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of POPL '89*, pp. 179–190. ACM, 1989.

[20] C. W. Probst, J. Hunker, and M. Bishop, editors. *Insider Threats in Cyber Security*, volume 49 of *Advances in Information Security*. Springer, 2010.

[21] H. Qunoo and M. Ryan. Modelling Dynamic Access Control Policies for Web-Based Collaborative Systems. In *Proc. of Conf. on Data and Applications Security and Privacy*, LNCS 6166, pp. 295-302, Springer, 2010.

[22] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.* 2:105–135, Feb. 1999.

[23] R. Sandhu and Q. Munawer. The ARBAC99 model for administration of roles. In *Proc. of ACSAC'99*, pp. 229–238. IEEE, 2002.

[24] A. Sasturkar, P. Yang, S. D. Stoller, and C. R. Ramakrishnan. Policy Analysis for Administrative Role Based Access Control. In *Proc. of Computer Security Foundations Workshop*, pp. 124-138. IEEE, 2006.

[25] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM* 32:733–749, July 1985.

[26] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable safety properties. *IEEE Symposium on Security and Privacy*, pp. 56-67, 2004.

[27] S. D. Stoller, P. Yang, M. Gofman, and C. R. Ramakrishnan. Symbolic reachability analysis for parameterized administrative role based access control. In *Proc. of SACMAT '09*, pp. 165–174. ACM, 2009.

[28] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *Proc. of CCS '07*, pp. 445–455. ACM, 2007.

[29] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.* 13:40:1–40:35, December 2010.

[30] N. Zhang, M. Ryan, and D. Guelev. Evaluating access control policies through model-checking. In *8th Information Security Conference*. Springer, 2005.

[31] N. Zhang, M. Ryan, and D. P. Guelev. Synthesizing verified access control systems through model checking. *J. of Computer Security* 16(1):1-61, 2008.