Term Rewriting for Access Control

Steve Barker and Maribel Fernández

King's College London,
Dept. of Computer Science,
Strand, London WC2R 2LS, U.K.

Abstract. We demonstrate how access control models and policies can be represented by using term rewriting systems, and how rewriting may be used for evaluating access requests and for proving properties of an access control policy. We focus on two kinds of access control models: discretionary models, based on access control lists (ACLs), and role-based access control (RBAC) models. For RBAC models, we show that we can specify several variants, including models with role hierarchies, and constraints and support for security administrator review querying.

1 Introduction

Access control has long been recognised as being of fundamental importance in computer security. In early work on access control models, Lampson [27] described the use of a matrix for describing the access privileges that users may exercise on system resources. Variations of the access matrix, typically *Access Control Lists (ACLs)*, are still very much in use today (see, for example, [16]). In recent years, *Role-Based Access Control (RBAC)* [32, 9] has emerged as *the* principal form of access control model in theory and practice.

For all types of access control models, from the access matrix to RBAC, researchers have recognised the importance of applying formal techniques to define access control models, access policies and the operational methods used for access request evaluation. Formal specification makes it possible to, for instance, compare policies rigorously, to understand the consequences of modifying policies, and to prove properties of policies.

In this paper, we demonstrate how *term rewriting* [15, 24, 5] may be profitably used in the formalisation of ACL and RBAC models and policies, and we demonstrate the use of rewriting for access request evaluation with respect to policies that are defined in terms of these models.

Term rewriting systems are usually defined by specifying a set of terms, and a set of rewrite rules that are used to "reduce" terms. This simple idea is very powerful: term rewriting techniques have been successfully applied to many domains in the last 20 years. They have had deep influence in the development of computational models, programming and specification languages, theorem provers and proof assistants. More recently, rewriting techniques have been fruitfully exploited in the context of security protocols (see, for instance, [10]) and security policies for controlling information leakage (see, for example, [17]).

Although rewriting is widely applicable, its application to problems in access control has hitherto been quite restricted (albeit [26], which uses graph transformations, is a notable exception). Instead, the emphasis in the literature, on the formalisation of access control, has been on the use of logic languages for (i) the specification of access control requirements [9, 22, 12], and (ii) sound, complete and PTIME operational methods for evaluating access control requests with respect to policy requirements (see, for example, [8]). Nevertheless, there are several reasons to consider the use of term rewriting approaches for ACL and RBAC model definition, policy specification and access request checking. The expressivity of term rewriting is an important reason for applying rewrite techniques to access control: in the past, rewriting systems have been used to specify, in a uniform way, several computational paradigms, including functional, logic, imperative and concurrent ones (see, for example, [4, 18, 21]); in this paper we will show that rewriting can also be used to define ACL and RBAC policies in a uniform and formal way. Another important reason to use rewrite-based languages to specify access control policies is that we can then apply rewriting techniques, and use tools such as ELAN [13, 23], MAUDE [14] and CiME (www.lri.fr), to study properties of the policies (for instance, to check confluence and termination of the reduction relation induced by the rewrite rules), to test, compare and experiment with evaluation strategies, to automate equational reasoning, and also for rapid prototyping of access policies. Rewriting systems can provide a formal basis for the study of a broad range of security issues (e.g., authentication [1, 20] and intrusion detection [2]). In this paper we will use term rewriting systems for the specification, implementation and validation of ACL and RBAC policies that are used to protect resources in centralised computer systems from pre-authenticated system users.

The rest of this paper is organised as follows. In Section 2, some preliminary notions are briefly described. Discretionary access control models are studied in Section 3, where we show how to specify ACLs as rewrite systems and how properties of ACL policies may be proven. In Section 4, we describe a variety of RBAC policies as rewrite systems, and we demonstrate how properties of these policies may be proven. In Section 5, we discuss related work. Finally, in Section 6, we draw conclusions and make suggestions for further work.

2 Preliminaries

We begin by describing the principal components of ACLs and RBAC. We then describe some basic notions on term rewriting. We refer the reader to [16, 9, 5] for additional information on ACLs, RBAC and term rewriting, respectively.

2.1 The Access Matrix and Access Control Lists

The language of the access matrix [27] includes a finite set $\mathcal{U}$ of *users* (e.g., human users and software agents), a finite set $\mathcal{O}$ of *objects* (e.g., files and directories), and a finite set $\mathcal{A}$ of *access privileges* (e.g., read, write and execute privileges).

The access matrix [27] itself includes a row for each subject, and a column for each object in the system. Each cell of the matrix describes the set of access privileges that a subject may exercise on an object. An access matrix is usually implemented as an ACL, which records for each subject the privileges on objects that are assigned to the subject. A *reference monitor* is used to evaluate requests by subjects to exercise access privileges on an object. A user $u \in \mathcal{U}$ is authorised to exercise an access privilege $p \in \mathcal{P}$ on an object $o \in \mathcal{O}$ if and only if the access matrix/access control list includes an entry that specifies that u is assigned the p privilege on o.

2.2 Role-based Access Control

In very simple terms, the fundamental idea of RBAC is that:

- a user u of a resource o may be assigned to a set of roles $\{r_1, \ldots, r_n\}$ (usually as a consequence of the user performing a job function in an organisation e.g., *doctor*, CEO, etc);
- access privileges on resources are also assigned to roles;
- a user u may exercise an access privilege p on a resource o if and only if u is assigned to a role r to which the privilege p on o is also assigned.

It follows, from the discussion above, that RBAC models/policies are specified with respect to a domain of discourse that includes the sets $\mathcal{U}$ of users, $\mathcal{O}$ of objects, and $\mathcal{P}$ of access privileges, together with a (finite) set $\mathcal{R}$ of *roles*.

The capability of assigning users to roles and permissions (i.e., access privilege assignments on objects) to roles are primitive requirements of all RBAC models. The most basic category of RBAC model, flat RBAC [32] (or $RBAC_F$ for short), requires that these types of assignment are supported. The $RBAC_{H2A}$ model extends $RBAC_F$ to include the notion of an RBAC role hierarchy (see below) in addition to user-role and permission-role assignments. The $RBAC_{C3A}$ model extends $RBAC_{H2A}$ by allowing constraints on policies to be represented, and the $RBAC_{S4A}$ model extends $RBAC_{C3A}$ by allowing administrator queries to be evaluated with respect to an RBAC policy specification. The flat RBAC, $RBAC_{H2A}$, $RBAC_{C3A}$ and $RBAC_{S4A}$ models from [32] are referred to, respectively, as $RBAC_F, RBAC^P_{H2A}, RBAC^P_{C3A}$ and $RBAC^P_{S4A}$ logic theories in the formal representation of RBAC models in [9]. In the remainder of the paper, we will refer to $RBAC_F, RBAC^P_{H2A}, RBAC^P_{C3A}$ and $RBAC^P_{S4A}$ theories rather than models.

In the $RBAC^P_{H2A}$ theory, the semantics of user-role assignment may be defined in terms of a 2-place *ura* predicate (where *ura* is short for "user role assignment") and permission-role assignment can be defined in terms of a 3-place predicate *pra* (where *pra* is short for "permission role assignment"). The extensions of these predicates define role and permission assignments in a world of interest.

Definition 1. *Let Π be an $RBAC^P_{H2A}$ theory. Then,*

- $\Pi \models ura(u, r)$ *if and only if user* $u \in \mathcal{U}$ *is assigned to role* $r \in \mathcal{R}$*;*
- $\Pi \models pra(a, o, r)$ *if and only if the access privilege* $a \in \mathcal{A}$ *on object* $o \in \mathcal{O}$ *is assigned to the role* $r \in \mathcal{R}$*.*

An $RBAC^P_{H2A}$ *role hierarchy* is defined as a (partially) ordered (and finite) set of roles. The ordering relation is a role seniority relation. In an $RBAC^P_{H2A}$ theory Π, a 2-place predicate $senior\_to(r_i, r_j)$ is used to define the seniority ordering between pairs of roles i.e., the role $r_i \in \mathcal{R}$ is a more senior role (or more powerful role) than role $r_j \in \mathcal{R}$. If r_i is senior to r_j then any user assigned to the role r_i has at least the permissions that users assigned to role r_j have. Role hierarchies are important for specifying implicitly the inheritance of access privileges on resources.

The semantics of the *senior_to* relation may be expressed, in terms of an $RBAC^P_{H2A}$ theory Π, thus:

- $\Pi \models senior\_to(r_i, r_j)$ if and only if the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy.

The *senior_to* relation may be defined as the reflexive-transitive closure of an irreflexive-intransitive binary relation *ds* (where *ds* is short for "directly senior to"). The semantics of *ds* may be expressed, in terms of an $RBAC^P_{H2A}$ theory Π, thus:

- $\Pi \models ds(r_i, r_j)$ iff $r_i \neq r_j$, the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy defined in Π, and there is no role $r_k \in \mathcal{R}$ such that $[ds(r_k, r_j) \wedge ds(r_i, r_k)]$ holds where $r_k \neq r_i$ and $r_k \neq r_j$.

Remark 1. In RBAC, users activate and deactivate roles in the course of session management. Session management is an implementation issue, the details of which will be the subject of future work.

Example 1. Suppose that the users u_1 *and* u_2 *are assigned to the roles* r_2 *and* r_1 *respectively, and that write (w) permission on object* o_1 *is assigned to* r_1 *and read (r) permission on* o_1 *is assigned to* r_2*. Moreover, suppose that* r_1 *is directly senior to* r_2 *in an* $RBAC^P_{H2A}$ *role hierarchy. Then, using the notation introduced above, this* $RBAC_{H2A}$ *policy is represented by the relations:*

$$ura(u_1, r_2), ura(u_2, r_1), pra(w, o_1, r_1), pra(r, o_1, r_2), ds(r_1, r_2).$$

User-role and permission-role assignments are related via the notion of an *authorisation*. An authorisation is a triple (u, a, o) that expresses that the user u has the a access privilege on the object o. Given an $RBAC^P_{H2A}$ theory Π, the set of authorisations $\mathcal{AUTH}$ defined by Π may be expressed thus:

$$(u, a, o) \in \mathcal{AUTH} \Leftrightarrow \exists r_1, r_2.ura(u, r_1) \ \wedge \ senior\_to(r_1, r_2) \ \wedge \ pra(a, o, r_2)$$

According to the definition of the set $\mathcal{AUTH}$ above, a user u may exercise the a access privilege on object o if:

u is assigned to the role r_1,[1] *r_1 is senior to a role r_2 in an $RBAC^P_{H2A}$ role hierarchy, and r_2 has been assigned the a access privilege on o.*

Example 2. By inspection of the user-role assignments, permission-role assignments, and the role seniority relationships that are specified in Example 1, it follows that the set of authorisations that are included in $\mathcal{AUTH}$ is:

$$\{(u_2, w, o_1), (u_2, r, o_1), (u_1, r, o_1)\}.$$

To extend $RBAC_{H2A}$ theories to $RBAC_{C3A}$ theories, *separation of duties constraints* must be supported. The *static separation of duties (ssd)* constraint is used to specify that a user cannot be assigned to a pair of mutually exclusive roles [9]. The *dynamic separation of duties (dsd)* constraint is used to prevent a user simultaneously activating a pair of roles that are specified as being dynamically separated [9].

To extend $RBAC_{C3A}$ programs to $RBAC_{S4A}$ theories, *permission-role review* must be possible in addition to *user-role reviews*, the latter being a requirement of $RBAC_F$ theories (see [32, 9]). That is, it must be possible for security administrators to pose queries on RBAC policy specifications to determine (i) the set of roles a user is assigned to, and (ii) the permissions that are assigned to roles.

2.3 Term Rewriting

Term rewriting systems can be seen as programming or specification languages, or as formulae manipulating systems that can be used in various applications such as operational-semantics specification, program optimisation or automated theorem proving. We recall briefly the definition of first-order terms and term rewriting systems, and refer the reader to [5] for further details and examples.

A *signature* $\mathcal{F}$ is a finite set of *function symbols* together with their (fixed) arity. $\mathcal{X}$ denotes a denumerable set of *variables*, and $T(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built up from $\mathcal{F}$ and $\mathcal{X}$.

Terms are identified with finite labeled trees, as usual. The symbol at the root of t is denoted by $root(t)$. *Positions* are strings of positive integers. The *subterm* of t at position p is denoted by $t|_p$ and the result of replacing $t|_p$ with u at position p in t is denoted by $t[u]_p$.

$\mathcal{V}(t)$ denotes the set of variables occurring in t. A term is *linear* if variables in $\mathcal{V}(t)$ occur at most once in t. A term is *ground* if $\mathcal{V}(t) = \emptyset$. Substitutions are written as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where t_i is assumed different from x_i. We use Greek letters for substitutions and postfix notation for their application.

Definition 2. *Given a signature $\mathcal{F}$, a* term rewriting system *on $\mathcal{F}$ is a set of rewrite rules $R = \{l_i \to r_i\}_{i \in I}$, where $l_i, r_i \in T(\mathcal{F}, \mathcal{X})$, $l_i \notin \mathcal{X}$, and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. A term t* rewrites *to a term u at position p with the rule $l \to r$ and the substitution σ, written $t \to_p^{l \to r} u$, or simply $t \to_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. Such a term t is called* reducible. *Irreducible terms are said to be in* normal form.

[1] Here we assume that u is also active in r_1 at the time of any access request.

We denote by $\to_R^+$ (resp. $\to_R^*$) the transitive (resp. transitive and reflexive) closure of the rewrite relation $\to_R$. The subindex R will be omitted when it is clear from the context.

Example 3. Consider a signature for lists of natural numbers, with function symbols:

- Z *(with arity 0) and* S *(with arity 1, denoting the successor function) to build numbers;*
- nil *(with arity 0, to denote an empty list),* cons *(with arity 2, to construct non-empty lists), and* append *(also with arity 2, to represent the operation that concatenates two lists).*

We can specify list concatenation with the following rewrite rules:

$$\mathsf{append}(\mathsf{nil}, x) \to x$$
$$\mathsf{append}(\mathsf{cons}(y, x), z) \to \mathsf{cons}(y, \mathsf{append}(x, z))$$

Then we have a reduction sequence:

$$\mathsf{append}(\mathsf{cons}(\mathsf{Z}, \mathsf{nil}), \mathsf{cons}(\mathsf{S}(\mathsf{Z}), \mathsf{nil})) \to^* \mathsf{cons}(\mathsf{Z}, \mathsf{cons}(\mathsf{S}(\mathsf{Z}), \mathsf{nil}))$$

Let $l \to r$ and $s \to t$ be two rewrite rules (we assume that the variables of $s \to t$ were renamed so that there is no common variable with $l \to r$), p the position of a non-variable subterm of s, and μ a most general unifier of $s|_p$ and l. Then $(t\mu, s\mu[r\mu]_p)$ is a *critical pair* formed from those rules. Note that $s \to t$ may be a renamed version of $l \to r$. In this case a superposition at the root position is not considered a critical pair.

A term rewriting system R is:

- *confluent* if for all terms t, u, v: $t \to^* u$ and $t \to^* v$ implies $u \to^* s$ and $v \to^* s$, for some s;
- *terminating* (or *strongly normalising*) if all reduction sequences are finite;
- *left-linear* if all left-hand sides of rules in R are linear;
- *non-overlapping* if there are no critical pairs;
- *orthogonal* if it is left-linear and non-overlapping;
- *non-duplicating* if for all $l \to r \in R$ and $x \in \mathcal{V}(l)$, the number of occurrences of x in r is less than or equal to the number of occurrences of x in l.

For example, the rewrite system in Example 3 is confluent, terminating, left-linear and non-overlapping (therefore orthogonal), and non-duplicating.

A *hierarchical union* of rewrite systems consists of a set of rules defining some basic functions (this is called the *basis* of the hierarchy) and a series of *enrichments*. Each enrichment defines a new function or functions, using the ones previously defined. Constructors may be shared between the basis and the enrichments.

We recall a modularity result for termination of hierarchical unions from [19] (Theorem 14), which will be useful later:

If in a hierarchical union the basis is non-duplicating and terminating, and each enrichment satisfies a general scheme of recursion, where each recursive call in the right-hand side of a rule uses subterms of the left-hand side, then the hierarchical union is terminating.

3 Access Control Lists as a Rewrite System

In this section, we illustrate the use of rewriting systems to specify ACL policies with an example. We do not claim that this is the only way to formalise an ACL policy as a rewrite system. Instead, our goal is to give an executable[2] specification of an ACL policy, to show some basic properties, and to address, using rewriting techniques, the problem of checking that the specification is consistent, correct, and complete (that is, no access can be both granted and denied, no unauthorised access is granted and no authorised access is denied).

3.1 Rewrite Rules

Consider a set of objects, and a set of user-identifiers: $\mathcal{U} = \{u_1, \ldots, u_n\}$, such that each user has a certain number of access privileges on those objects. For simplicity, assume that user identifiers are natural numbers, and to make the example more concrete, assume that the objects are files and the access privileges are read (r), write (w) or execute (x). For simplicity, we will only consider one file (the generalisation to many files is straightforward). The policy that we will model specifies that a user with an even identifier has rw rights (i.e., can read and write on the file), whereas users with odd numbers can only read, and users whose identifier is a multiple of 4 can read, write and also execute the file.

Users will request access to the file by using the function access, which will grant or deny the access depending on the user and the operation requested. Requests will be expressed as $\mathsf{access}(u, req)$ where u is a user-identifier and req is either r, w or x. The request will be evaluated using the rewrite system R_{ACL} given below; the result will be either grant or deny.

In the rewrite rules below, we denote variables with capital letters (e.g., U is a variable), and function symbols (including constants) with lower-case letters (e.g. r, w, x are constants). We use $\mathsf{rem}(n, m)$ to compute the remainder of the division of n by m.

$$\mathsf{access}(U, R) \rightarrow \mathsf{acl}(\mathsf{rem}(U, 2), R, U)$$

$$\begin{aligned} \mathsf{acl}(1, r, U) &\rightarrow \mathsf{grant} \\ \mathsf{acl}(1, w, U) &\rightarrow \mathsf{deny} \\ \mathsf{acl}(1, x, U) &\rightarrow \mathsf{deny} \end{aligned}$$

$$\begin{aligned} \mathsf{acl}(0, r, U) &\rightarrow \mathsf{grant} \\ \mathsf{acl}(0, w, U) &\rightarrow \mathsf{grant} \\ \mathsf{acl}(0, x, U) &\rightarrow f(\mathsf{rem}(U, 4)) \end{aligned}$$

$$\begin{aligned} f(0) &\rightarrow \mathsf{grant} \\ f(1) &\rightarrow \mathsf{deny} \\ f(2) &\rightarrow \mathsf{deny} \\ f(3) &\rightarrow \mathsf{deny} \end{aligned}$$

[2] For instance, the language MAUDE [14] can be used to execute rewrite-based specifications

For example, with these rewrite rules a request from user 101 to write on the file is denied, whereas a request from user 20 to execute it is granted, since:

$$\mathsf{access}(101, w) \rightarrow^*_{R_{ACL}} \mathsf{deny}.$$
$$\mathsf{access}(20, x) \rightarrow^*_{R_{ACL}} \mathsf{grant}.$$

R_{ACL} provides an executable specification of the policy (the rewrite rules are both a specification *and* an implementation of the access control function).

3.2 Properties of the Policy

In order for an access policy to be "acceptable", it is necessary that the policy satisfies certain acceptability criteria. As an informal example, it may be necessary to ensure that an access policy formulation does not specify that any user is granted and denied the same access privilege on the same data item (i.e., that the policy is consistent).

The following properties of R_{ACL} are easy to check, and will be used to prove that the policy specified is consistent, correct, and complete.

Property 1. The rewrite system R_{ACL} is terminating and confluent.

> *Proof.* Termination is trivially obtained, since R_{ACL} is a first-order system, and there are no recursive or mutually recursive functions.
> To prove confluence, first note that there are no critical pairs, therefore the system is locally confluent. Termination and local confluence imply confluence, by Newman's Lemma [30].

Corollary 1. *Every term has a unique normal form in R_{ACL}.*

As a consequence of the unicity of normal forms, our specification of the access control policy is *consistent*.

Property 2 (Consistency). For any user u and request req, it is not possible to derive both grant and deny for a request $\mathsf{access}(u, req)$.

We can give a characterisation of the normal forms:

Property 3. The normal form of a ground term of the form $\mathsf{access}(u, req)$ where u is a number and $req \in \{r, w, x\}$ is either grant or deny.

As a consequence, our specification of the access control policy is *total*, in the sense that any valid request (i.e., a request from a valid user to perform a valid operation on an existing object) produces a result (a denial or an acceptance).

Property 4 (Totality). Each access request $\mathsf{access}(u, req)$ from a valid user u to perform a valid operation req is either denied or granted.

Correctness and *Completeness* are also easy to check:

Property 5 (Correctness and Completeness). For any user u and request req:

– $\mathsf{access}(u, req) \rightarrow^* \mathsf{grant}$ if and only if u has the access privilege req on the file.
– $\mathsf{access}(u, req) \rightarrow^* \mathsf{deny}$ if and only if u does not have the access privilege req on the file.

Proof. Since we have consistency and totality, it is sufficient to show:

$$\mathsf{access}(u, req) \rightarrow^* \mathsf{grant} \text{ if and only if } u \text{ has the access privilege } req.$$

This is shown by inspection of the rewrite rules.

4 RBAC Policy Specifications as Rewrite Systems

In this section, we specify RBAC policies in terms of a rewrite system that is confluent and terminating. The normal forms of access requests are grant/deny i.e., each access request is reducible to grant or deny but not both.

4.1 Rewrite Rules

As indicated in Section 2, RBAC policies are specified with respect to: a set $\mathcal{U}$ of users, a set $\mathcal{O}$ of objects, a set $\mathcal{A}$ of access privileges, and a set $\mathcal{R}$ of roles.

We will use the function $\mathsf{roles} : \mathcal{U} \rightarrow List(\mathcal{R})$ to represent the assignment of roles to users (note that a user may be assigned to several roles). Lists will be built from constructors nil and cons (see Example 3), and we will write $[e_1, \ldots, e_n]$ as an abbreviation for the list constructed from the elements $e_1, \ldots, e_n$. It is worth noting that a predicate ura, as discussed in Section 2, can also be specified with rewrite rules, since predicates are boolean functions (in this case we could define a function from pairs $(u, r) \in (\mathcal{U} \times \mathcal{R})$ to booleans $\mathsf{True}, \mathsf{False}$). However, we prefer to model ura as a function from users to lists of roles because of the additional advantages this provides. In particular, modelling ura as a function makes it easy to obtain all of the roles that a specific user is assigned to. This is an essential requirement of $RBAC_F$ theories, which emphasise the importance of performing administrative checks of user-role assignments from an RBAC policy specification. The following rewrite rules specify a function roles, where we assume $\mathcal{U} = \{u_1, \ldots, u_n\}$ and each $r_{ij} \in \mathcal{R}$.

$$\begin{aligned} \mathsf{roles}(u_1) &\rightarrow [r_{11}, \ldots, r_{1i}] \\ &\vdots \\ \mathsf{roles}(u_n) &\rightarrow [r_{n1}, \ldots, r_{nk}] \end{aligned}$$

To represent the assignment of privileges to roles (called pra in Section 2), we have again two design choices: we could use a boolean function (i.e., , a predicate) with three arguments (role, access privilege, object) or we can use a function priv from roles to lists of pairs $(a, o) \in (\mathcal{A} \times \mathcal{O})$, $\mathsf{priv} : \mathcal{R} \rightarrow List(\mathcal{A} \times \mathcal{O})$. The second approach has advantages from a security administrator's point of view, since a function priv, to compute the set of access privileges assigned to

a role, can be used to perform checks on the access policy specification (as required for $RBAC^{P}_{S4A}$ policies). We define priv by the following set of rules, where $r_i, \ldots, r_n \in \mathcal{R}$, $a_{ij} \in \mathcal{A}$, and $o_{ij} \in \mathcal{O}$.

$$\begin{aligned} \mathsf{priv}(r_1) &\rightarrow [(a_{11}, o_{11}), \ldots, (a_{1i}, o_{1i})] \\ &\vdots \\ \mathsf{priv}(r_n) &\rightarrow [(a_{n1}, o_{n1}), \ldots, (a_{nk}, o_{nk})] \end{aligned}$$

Example 4. The user-role and permission-role assignments described in Example 1 may be expressed by the following rewrite rules:

$$\begin{aligned} \mathsf{roles}(u_1) &\rightarrow [r_2] \\ \mathsf{roles}(u_2) &\rightarrow [r_1] \\ \mathsf{priv}(r_1) &\rightarrow [(w, o_1)] \\ \mathsf{priv}(r_2) &\rightarrow [(r, o_1)] \end{aligned}$$

Access requests from users can be evaluated by using a rewrite system to grant or deny the request according to the user-role and permission-role assignments that are included in an RBAC policy specification. For that, we may use the following rules, where U, A, O, R, L are variables and the operators member and $\cup$ are the standard membership test and union operators.

$$\begin{aligned} \mathsf{access}(U, A, O) &\rightarrow \mathsf{check}(\mathsf{member}((A, O), \mathsf{privileges}(\mathsf{roles}(U)))) \\ \mathsf{check}(\mathsf{True}) &\rightarrow \mathsf{grant} \\ \mathsf{check}(\mathsf{False}) &\rightarrow \mathsf{deny} \\ \\ \mathsf{privileges}(\mathsf{nil}) &\rightarrow \mathsf{nil} \\ \mathsf{privileges}(\mathsf{cons}(R, L)) &\rightarrow \mathsf{priv}(R) \cup \mathsf{privileges}(L) \end{aligned}$$

For example, with the assignment shown in Example 4, we have a reduction sequence: $\mathsf{access}(u_1, r, o_1) \rightarrow^* \mathsf{grant}$.

In the discussion that follows, we will use R_{RBAC} to refer to the rewrite system that contains the set of rules that we have defined in this section.

4.2 Properties of the RBAC Policy

The following properties of R_{RBAC} are easy to check and will be used to show that the specification is consistent, correct and complete:

Property 6. The rewrite system R_{RBAC} is terminating and confluent.

Proof. To prove termination, we use a modularity result for hierarchical unions (see Section 2 and [19]). First, observe that the system R_{RBAC} is hierarchical: the rules defining roles, priv and check form the basis of the hierarchy, they are trivially terminating since the right-hand sides of rules are normal forms, and they are non-duplicating because the right-hand sides contain no variables. The rules defining $\mathsf{privileges}$ are recursive, but the recursive call is made on a subterm of the left-hand side argument. The rule

defining access is not recursive. Therefore, the rules defining privileges and access satisfy the recursive scheme and the full system is terminating.
To prove confluence, first note that there are no critical pairs, therefore the system is locally confluent. Termination and local confluence imply confluence, by Newman's Lemma [30].

Corollary 2. *Every term has a unique normal form in R_{RBAC}.*

As a consequence of the unicity of normal forms, our specification of the RBAC policy R_{RBAC} is *consistent*.

Property 7 (Consistency). For any $u \in \mathcal{U}$, $a \in \mathcal{A}$, $o \in \mathcal{O}$: it is not possible to derive, from R_{RBAC}, both grant and deny for a request $\mathsf{access}(u, a, o)$.

We can give a characterisation of the normal forms:

Property 8. The normal form of a ground term of the form $\mathsf{access}(u, a, o)$ where $u \in \mathcal{U}$, $a \in \mathcal{A}$ and $o \in \mathcal{O}$ is either grant or deny.

As a consequence, our specification of the access control policy is *total*.

Property 9 (Totality). Each access request $\mathsf{access}(u, a, o)$ from a valid user u to perform a valid action a on the object o is either granted or denied.

Correctness and *Completeness* are also easy to check:

Property 10 (Correctness and Completeness). For any $u \in \mathcal{U}$, $a \in \mathcal{A}$, $o \in \mathcal{O}$:

- $\mathsf{access}(u, a, o) \rightarrow^* \mathsf{grant}$ if and only if u has the access privilege a on o.
- $\mathsf{access}(u, a, o) \rightarrow^* \mathsf{deny}$ if and only if u does not have the access privilege a on o.

Proof. Since the specification is consistent and total, it is sufficient to show that $\mathsf{access}(u, a, o) \rightarrow^* \mathsf{grant}$ if and only if u is assigned the access privilege a on the object o. By inspection of the rewrite rules:

$$\mathsf{access}(u, a, o) \rightarrow \mathsf{check}(\mathsf{member}(a, o), \mathsf{privileges}(\mathsf{roles}(u)))$$

Therefore, the result is grant if and only $(a, o) \in \mathsf{privileges}(\mathsf{roles}(u))$ if and only if $(a, o) \in \mathsf{priv}(r)$ for some $r \in \mathsf{roles}(u)$.

It is important to note that the proofs above do not have to be generated by a security administrator; rather, the proofs demonstrate that an RBAC policy R_{RBAC} satisfies the properties described above. A security administrator can simply base an RBAC policy on the term rewrite system that we have defined and can be sure that the properties of R_{RBAC} hold.

4.3 RBAC with a Hierarchy of Roles: $RBAC^{P}_{H2A}$ Policies

It is easy to accommodate a notion of seniority of roles where a role inherits, via a role hierarchy, the privileges of its subordinate roles (as explained in Section 2). For that, we just add rules of the form $\mathsf{dsub}(r_i) \rightarrow [r_1, \ldots, r_j]$ to specify a function $\mathsf{dsub} : \mathcal{R} \rightarrow \mathit{List}(\mathcal{R})$, where $\mathsf{dsub}(r_i) = [r_1, \ldots, r_j]$ means that $r_1, \ldots, r_j$ are direct subordinate roles of r_i (hence r_i is directly senior to $r_1 \ldots r_n$). Then, we redefine the privileges of a role as its privileges plus the privileges of its direct subordinate roles. We use the functions dp to compute direct privileges (which corresponds to the previously defined priv) and the function $\mathsf{privileges}$ defined above:

$$\mathsf{priv}(r) \rightarrow \mathsf{dp}(r) \cup \mathsf{privileges}(\mathsf{dsub}(r))$$

Note that we do not need to change the definition of access (see Section 4.1) to accommodate hierarchies of roles, and we do not need to impose conditions on the form that a role hierarchy takes (apart from an acyclicity condition, which is a natural requirement for RBAC role hierarchies).

There are obvious optimisations that could be made if the hierarchy contains sharing (i.e., we should avoid computing twice the privileges of a role if it appears as a subordinate role of several of a user's roles). For instance, we may want to compute first all the roles of a user, including subordinate ones, and then the privileges of this set of roles. Efficiency considerations will be addressed in future work.

4.4 RBAC with constraints and reviews

For RBAC policies beyond $RBAC^{P}_{H2A}$ policies, separation of duties constraints must be supported and it must be possible for security administrators to review policy specifications (beyond simple user-role reviewing). We can implement several administrative checks on an RBAC policy, again as rewrite rules.

Separation of Duties is the property that specifies that roles assigned to a user cannot be mutually exclusive. To ensure that a specification of an RBAC policy satisfies the separation of duties property, we will erase conflicting roles assigned to a user (producing a list of roles without mutually exclusive pairs). This is obtained by evaluation of $\mathsf{clean}(\mathsf{roles}(u))$ in a rewrite system containing the rules:

$$\begin{aligned}
\mathsf{clean}(\mathsf{nil}) &\rightarrow \mathsf{nil} \\
\mathsf{clean}(\mathsf{cons}(R, L)) &\rightarrow \mathsf{cons}(R, \mathsf{clean}(\mathsf{eraseclash}(R, L))) \\
\mathsf{eraseclash}(R, \mathsf{nil}) &\rightarrow \mathsf{nil} \\
\mathsf{eraseclash}(R, \mathsf{cons}(R', L)) &\rightarrow \mathsf{cons}(R', \mathsf{eraseclash}(R, L)) \qquad (R, R' \text{ do not clash}) \\
\mathsf{eraseclash}(R, \mathsf{cons}(R', L)) &\rightarrow \mathsf{eraseclash}(L) \qquad (R, R' \text{ clash})
\end{aligned}$$

Reviews We can add to the specification R_{RBAC} the rules given below. Then, to check that every user has been assigned a role, an administrator could simply

evaluate the term $\mathsf{RolesDefined?}(u)$.

$$\begin{aligned}\mathsf{RolesDefined?}(u) &\rightarrow \mathsf{review}(\mathsf{roles}(u))\\ \mathsf{review}(\mathsf{nil}) &\rightarrow \text{“error: user without a role”}\\ \mathsf{review}(\mathsf{cons}(r, lr)) &\rightarrow \text{“OK”}\end{aligned}$$

5 Related Work

In terms of security applications, we note that the SPI-calculus [1] was developed as an extension of the π-calculus for proving the correctness of authentication protocols. In [3], the π-calculus is applied to reason about a number of basic access control policies and access mechanisms. However, the work described in [3] does not treat RBAC models and policies as rewrite systems. The work most closely related to ours is Koch et al's proposal [26]. In [26], RBAC is formalised by using a graph-based approach, with graph transformation rules used for describing the effects of actions as they relate to RBAC notions. This formalisation is used by Koch et al as a basis for proving properties of RBAC specifications, based on the categorical semantics of the graph transformations. Our work addresses similar issues to Koch et al's work but provides a different formulation of RBAC policies, and focuses on operational aspects. We use rewrite rules both as a specification and an implementation of an access control policy. To obtain efficient evaluators for request evaluation, sharing of computations is an important issue; for that, we note that graph-based rewriting may be used to devise efficient evaluation strategies.

In recent years, researchers have developed some sophisticated access control models in which access control requirements may be expressed by using rules that are employed to reason about authorised forms of access (see, for example, [22], [11], and [9]). In these approaches, the requirements that must be satisfied in order to access resources are specified by using rules expressed in (C)LP languages and access request evaluation may be viewed as being performed by rewriting, using, for example, SLG-resolution [33] or constraint solvers [28]. Our term rewriting approach offers similar attractions to the (C)LP approaches. We envisage term rewriting, or more generally, equational specifications, being used as an alternative to (C)LP. Term rewriting offers an algebraic approach to specification, where functional definitions can be easily accommodated. In this paper we have used first-order rewriting system; we could also consider a more restricted framework, for instance, orthogonal rewrite systems (in which case the confluence property is guaranteed). On the other hand, we could also consider more general rewriting frameworks, such as higher-order rewriting systems [25, 29], to gain expressivity: we could then use higher-order functions in our policy specifications.

6 Conclusions and Further Work

In this paper, we have described the representation of ACL policies and RBAC policies as term rewrite systems. In particular, we have shown how different ac-

cess control models may be flexibly defined in a completely uniform way. We also demonstrated how access requests may be evaluated with respect to an access policy specification by term rewriting, and how (static) properties of policies may be proven of ACL and RBAC policy specifications.

We have argued that term rewriting is particularly attractive in allowing multiple access control models and policies to be defined in a uniform way. In future work, we intend to consider the use of term rewriting for the specification of access control models other than ACL and RBAC. In particular, we wish to consider the specification of usage control models [31] as term rewrite systems, and access control models that may be used in a distributed computing environments. We also intend to apply our term rewriting approach to problems relating to the administration of RBAC policies (e.g., issues of administrative delegation), and to the specification of RBAC policies that allow conditional user-role, permission-role and denial-role assignments to be specified (see, for example, [6] and [9]). We also propose to investigate the use of policy materialisation [22] and policy specialisation methods [8] for the optimisation of access request evaluation with respect to the formulation of ACL and RBAC policies as rewrite systems.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conf. on Computer and Communication Security*, pages 36–47, 1997.
2. T. Abbes, A. Bouhoula, and M. Rusinowitch. Protocol analysis in intrusion detection using decision tree. In *Proc. ITCC'04*, pages 404–408, 2004.
3. J. Abendroth and C. Jensen. A unified security mechanism for networked applications. In *SAC2003*, pages 351–357, 2003.
4. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 2004.
5. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
6. S. Barker. Data protection by logic programming. In *Proc. 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1300–1314. Springer-Verlag, 2000.
7. S. Barker. Protecting deductive databases from unauthorized retrieval and update requests. *Journal of Data and Knowledge Engineering*, 23(3):231–285, 2002.
8. S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via jones optimality logic program specialisation. *HOSC, To Appear*, 2006.
9. S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. on Information and System Security*, 6(4):501–546, 2003.
10. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: a toolset to reason about the JavaCard platform. In *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
11. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proc. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, 2002.

12. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *SACMAT*, pages 41–52, 2001.
13. P. Borovansky, C. Kirchner, H. Kirchner, and P-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
14. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, 2003.
15. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, volume B. North-Holland, 1989.
16. S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Access control: principles and solutions. *Softw., Pract. Exper.*, 33(5):397–421, 2003.
17. R. Echahed and F. Prost. Security policy in a declarative style. In *Proc. 7th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'05)*. ACM Press, 2005.
18. M. Fernández. *Programming Languages and Operational Semantics: An Introduction*. King's College Publications, 2004.
19. M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Recent Trends in Data Type Specification. Proc. 10th. Workshop on Specification of Abstract Data Types (ADT'94)*, number 906 in LNCS, 1995.
20. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4):341–383, 2004.
21. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, ACM Press, 1997.
22. S. Jajodia, P. Samarati, M. Sapino, and V.S. Subrahmaninan. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.
23. C. Kirchner, H. Kirchner, and M. Vittek. *ELAN user manual*. Nancy (France), 1995. Technical Report 95-R-342, CRIN.
24. J.-W. Klop. Term Rewriting Systems. In S. Abramsky, Dov.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
25. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
26. M. Koch, L. Mancini, and F. Parisi-Presicce. A graph based formalism for rbac. In *SACMAT*, pages 129–187, 2004.
27. Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.
28. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
29. Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.
30. M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.
31. J. Park and R. Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.
32. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. 4th ACM Workshop on Role-Based Access Control*, pages 47–61, 2000.
33. *The XSB System Version 2.7.1, Programmer's Manual*, 2005.