Dynamic Event-Based Access Control as Term Rewriting*

Clara Bertolissi[2], Maribel Fernández[1], and Steve Barker[1]

[1]King's College London, Dept. of Computer Science, London WC2R 2LS, U.K.
[2]LIF, Université de Provence, Marseille, France
{Steve.Barker, Clara.Bertolissi, Maribel.Fernandez}@kcl.ac.uk

Abstract. Despite the widespread adoption of Role-based Access Control (RBAC) models, new access control models are required for new applications for which RBAC may not be especially well suited and for which implementations of RBAC do not enable properties of access control policies to be adequately defined and proven. To address these issues, we propose a form of access control model that is based upon the key notion of an event. The access control model that we propose is intended to permit the representation of access control requirements in a distributed and changing computing environment, the proving of properties of access control policies defined in terms of our model, and direct implementations for access control checking.

1 Introduction

Included amongst the most important problems in access control are the problems of formally defining richly expressive access control models that enable security administrators to specify a wide range of policies, using declarative languages, to prove properties of access control policies (for assurance purposes), and to evaluate access requests efficiently with respect to a representation of an access control policy. The increased use of access control policies in distributed computing environments has increased the need to have formal access control policies that are declarative (to handle the complexities of policy specification), to prove properties of policies (for verifiability purposes), and for efficient evaluation (given the computational overheads of potentially accessing large volumes of data from multiple locations). Moreover, given the complexities and scope involved, distributed applications have increased the requirements for autonomous changing of access control policies.

In recent years, work on RBAC [28, 8] has emerged as *the* principal type of access control model in theory and in practice. RBAC is well suited for use with relatively static, closed, centralized systems where the assignment of a known, identifiable user to a role is specified by (typically) a centrally organized team of

* Research partially funded by the EU project *Implementing access control mechanisms using rewriting techniques*, Marie Curie Intra European Fellowships Programme.

human security administrators that has complete information about user qualifications and responsibilities, and hence user assignments to well-defined job functions and thus roles. Administrators are also (typically) assumed to have complete information about the permissions to be assigned to roles. The administrators of an RBAC policy for a centralized system will revise a policy formulation to take into account any changes to role and permission assignments (as required to meet organizational demands). These changes do not usually need to be performed in real-time; in general, RBAC policy specifications for centralized systems are relatively static (i.e., user-role, permission-role, and role-role relationships often persist for long periods of time). The features of RBAC have been selected to map onto organizational structures, and RBAC policies, in the centralized case, are mandatory in the sense that a user's access privileges on information sources is determined by the job function the user performs within the organization.

The features of RBAC that make it suitable for use in the centralized case are not necessarily so relevant in certain distributed computing contexts. In certain distributed environments, entities that request access to resources may not be known to enterprises with resources to protect (i.e., users may be stranger agents), *open* access policies are natural to adopt [8], as well as closed policies, decisions on access are more likely to need to be delegated to third-parties (e.g., in the case where the information about the identity or attributes of requesters may be required), the qualifications and responsibilities of requesters do not necessarily have any significance, in terms of access control, and the notion of a job function may not apply (as requesters for access to an enterprise's resources may have no connection with the enterprise). Although the features of RBAC naturally map to organizational structures, for many distributed applications the concept of an organizational structure may be irrelevant. The size, complexity and dynamic nature of some distributed systems present particular challenges that demand that changes to access policies be made frequently (e.g., in response to sales patterns and volumes) and by autonomous means (rather than by human security administrators manually modifying policy specifications). In the decentralized case, modifications to access policies for protecting an organization's assets may need to be made in response to events in external environments over which a policy administrator has no control, and about which administrators may not have complete information. Moreover, the high complexity of access control policies in the decentralized case demands not only that rich forms of language be used to represent these requirements but also that effective proof methods be employed to guarantee satisfaction of properties of policies.

To address the requirements for formal access policy representation for dynamic, distributed information systems, we propose an event-based distributed access control model, we demonstrate how and why access control policies, defined in terms of our model, should be considered as *term rewrite systems* [16, 23, 4], and we introduce *distributed term rewriting systems.* The model that we propose, and its representation using term rewriting, contributes to the literature on formal access control models by demonstrating how access control models may be defined that enable the autonomous changing of access control policies, the proving of properties of policies, and the efficient evaluation of access requests when the sources of access control and user requested information may be widely dispersed. We call the access control model that we introduce the *Dynamic Event-Based Access Control (*DEBAC*)* model. The *DEBAC* model addresses a number of limitations of RBAC when the latter is applied in distributed computational contexts: that in certain distributed environments, entities that request access to resources may not be known (so it is not possible to authorize access on the basis of a job function/role); that user authorizations may change dynamically on the basis of the occurrence of a wider range of events than the role and permission assignments used in RBAC; and that the information that is used to decide on granting/denying a user's request may be distributed across several sites (rather than being centrally located). We also demonstrate that the expressiveness of the *DEBAC* model and the representation of *DEBAC* policies, as term rewrite systems, permit a range of properties to be proven of *DEBAC* policies; these proofs guarantee that security goals are satisfied by a policy specification and by the operational methods used to evaluate access requests. Approaches that provide for provably correct security have always been and remain of high interest in the security community.

Represented as term rewrite systems, *DEBAC* policies are specified as a set of rules and access requests are specified as terms. Access request evaluation is effected by "reducing" terms to a *normal form* (see below). Term rewriting techniques have been successfully applied, and have had deep influence, in the development of computational models, programming and specification languages, theorem provers, and proof assistants. More recently, rewriting techniques have also been fruitfully exploited in the context of security protocols (see, for instance, [9]), and security policies for controlling information leakage (see, for example, [18]). As we will see, representing *DEBAC* policies as term rewriting systems enables complex and changing access control requirements to be succinctly specified in a declarative language that is formally well defined. The formal foundations on which our approach is based make it possible to apply the extensive theory of rewriting to access control; in particular, standard rewriting techniques can be used to show that access control policies satisfy essential properties (such as consistency and completeness) and can be used to study combinations of policy specifications. Another important reason to use rewrite-based languages to specify access control policies is that tools, such as ELAN [14, 22] and MAUDE [15], can be used to test, compare and experiment with access request evaluation strategies, to automate equational reasoning, and for the rapid prototyping of access control policies.

The rest of this paper is organized as follows. In Section 2, we give some details on term rewriting to help to make the paper self-contained. In Section 3, we describe the *DEBAC* model and we introduce distributed rewrite systems as a tool to define *DEBAC* policies. In Section 4 we show how *DEBAC* policies can be specified via rewrite rules and we use this specification for proving essential properties of these policies. Some extensions of *DEBAC* policies are investigated in Section 5. We discuss related work in Section 6. Finally, we draw conclusions and make suggestions for further work in Section 7.

2 Preliminaries

In this section we recall some basic notions and notations for term rewriting. We refer the reader to [4] for additional information.

Term rewriting systems can be seen as programming or specification languages, or as formulae manipulating systems. They have been used in various applications (e.g., operational semantics, program optimization, automated theorem proving, and recently, computer security). We recall briefly the definition of first-order terms and term rewriting systems.

A *signature* $\mathcal{F}$ is a finite set of *function symbols* together with their (fixed) arity. $\mathcal{X}$ denotes a denumerable set of *variables* $X_1, X_2, \ldots$, 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. 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 to be different from the variable X_i. We use Greek letters for substitutions and postfix notation for their application. We say that two terms unify if there is some substitution that makes them equal. Such a substitution is called a *unifier*. The *most general unifier* (mgu) is the unifier that will yield instances in the most general form.

Definition 1. *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.

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 1. 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),* head *and* tl *(with arity 1, to obtain the head and tail of a list, resp.), and* length *(also with arity 1, to compute the length of a list).*

The list containing the numbers 0 *and* 1 *is written:* $\mathsf{cons}(\mathsf{z}, \mathsf{cons}(\mathsf{s}(\mathsf{z}), \mathsf{nil}))$*, or simply* $[\mathsf{z}, \mathsf{s}(\mathsf{z})]$ *for short. We can specify the functions* head*,* tl *and* length *with rewrite rules as follows:*

$$\begin{aligned}\mathsf{head}(\mathsf{cons}(X, L)) &\to X\\ \mathsf{tl}(\mathsf{cons}(X, L)) &\to L\\ \mathsf{length}(\mathsf{nil}) &\to \mathsf{z}\\ \mathsf{length}(\mathsf{cons}(X, L)) &\to \mathsf{s}(\mathsf{length}(L))\end{aligned}$$

Then we have a reduction sequence:

$$\begin{aligned}&\mathsf{length}(\mathsf{cons}(\mathsf{z}, \mathsf{cons}(\mathsf{s}(\mathsf{z}), \mathsf{nil}))) \to \mathsf{s}(\mathsf{length}(\mathsf{cons}(\mathsf{s}(\mathsf{z}), \mathsf{nil})) \to\\ &\mathsf{s}(\mathsf{s}(\mathsf{length}(\mathsf{nil}))) \to \mathsf{s}(\mathsf{s}(\mathsf{z}))\end{aligned}$$

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 normalizing*) 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 R 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 1 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 The *DEBAC* model

In this section, we describe the principal components of the *DEBAC* model and introduce distributed term rewriting systems.

3.1 Features of *DEBAC* models

We begin by defining some of the key sets of constants in the signature that we use in the formulation of the *DEBAC* model and *DEBAC* policies. Specifically, we require:

- A countable set $\mathcal{R}$ of *resources*, written $r_1, r_2, \ldots$.
- A countable set $\mathcal{A}$ of named *actions* $a_1, a_2, \ldots$.
- A countable set $\mathcal{U}$ of *user identifiers*, written $u_1, u_2, \ldots$.
- A countable set $\mathcal{C}$ of *categories* $c_0, c_1, \ldots$.
- A countable set $\mathcal{E}$ of *event identifiers* $e_1, e_2, \ldots$.
- A countable set $\mathcal{S}$ of *site identifiers*, we use Greek letters $\mu, \nu, \ldots$ as site identifiers.
- A countable set $\mathcal{T}$ of *time points.*

The fundamental notion on which our *DEBAC* model is based is that of an event. In the *DEBAC* model, events are happenings at an instance of time that involve users performing actions. We view events as structured and described via a sequence l of ground terms of the form $\mathsf{event}(e_i, u, a, t)$ where event is a data constructor of arity four, e_i $(i \in \mathbb{N})$ are constants in $\mathcal{E}$ (denoting unique event identifiers), $u \in \mathcal{U}$ identifies a user, a is an action associated to the event, and t is the time when the event happened. In the discussion that follows, we represent times as natural numbers in $YYYYMMDD$ format, and we assume that time is bidirectional so that proactive and postactive changes may be made to represent access policy requirements, and past, present and future times can be used in our model to make access control decisions.

In the *DEBAC* model, users may request to perform actions on resources that are potentially accessible from any site in a distributed system. A user is assigned to a particular category (e.g., *normal users, preferred users*, etc) on the basis of the history of events that relate to the user. Access to resources is then defined in the following way:

> *A user $u \in \mathcal{U}$ is permitted to perform an action $a \in \mathcal{A}$ on a resource $r \in \mathcal{R}$ that is located at site $s \in \mathcal{S}$ if and only if u is assigned to a category $c \in \mathcal{C}$ to which a access on r has been assigned.*

In the *DEBAC* model, assignments of a user u to a category c are based on the occurrence of events that are recorded in the history of events relating to u. As we will see later, hierarchies of categories of users may also be naturally accommodated in the *DEBAC* model.

We formally specify the notion of permitted access in term rewriting form below. In a *DEBAC* specification, there are two kinds of functions, which we call *generic* and *specific*, respectively. Generic functions are common to all *DEBAC* specifications, whereas specific functions, as their name suggests, depend on the specific scenario that we are modeling.

Given an event e_i, we will use standard generic functions to extract the component information from an event description. For instance, we may define a function user that returns the user involved in a given event, as follows:

$$\mathsf{user}(\mathsf{event}(E, U, A, T)) \to U$$

We assume that events are atomic, however, our model could be generalized to permit the representation of events that take place over a period of time and events that are composed of atomic parts (sub-events).

Also, we have chosen to include the time as an explicit component of an event. In certain contexts, it is sufficient to know the order of events. In such cases, the position of the event in a list of events provides enough information and the time parameter may be omitted.

3.2 Distributed term rewriting systems

An important aspect of the *DEBAC* model is the capability of representing systems where resources may be distributed across different sites, and the information needed to decide whether a user request is granted or denied may also be distributed. To address this issue, we will define access control policies as modular term rewriting systems, where modules may be independently maintained at different sites, and information sources may be explicitly specified. In other words, policy designers may directly define the sites (locations) to be used in access request evaluation.

For the approach to distributed access control that we propose, we introduce distributed term rewriting systems (DTRSs); DTRSs are term rewriting systems where rules are partitioned into modules, each associated with a site, and function symbols are annotated with site identifiers. We assume that each site has a unique identifier (we use Greek letters $\mu, \nu, \ldots$ to denote site identifiers).

We say that a rule $f(t_1, \ldots, t_n) \to r$ defines f. There may be several rules defining f; we will assume that they are all at the same site ν. We write f_ν to indicate that the definition of the function symbol f is stored in the site ν. If a symbol is used in a rule without a site annotation, we assume the function is defined locally.

For example, in a DTRS used in a bank scenario, we may have a local function $\mathsf{account}$ such that $\mathsf{account}(u)$ returns u's bank account number, and rules computing the average balance of a user's account, stored in a site ν. Then we could define the security category of a user u using a rule

$$\begin{aligned} \mathsf{category}(U) \to\ & if\ \mathsf{averagebalance}_\nu(\mathsf{account}(U)) \geq 10000 \\ & then\ \mathsf{VIP\ CLIENT} \\ & else\ \mathsf{NORMAL\ CLIENT} \end{aligned}$$

The example above describes one instance of rule specification to illustrate the use of annotations on function symbols (we redefine categories for users in a more general context in the next section). Here, to calculate u's security category as a client, the average balance of u's account has to be computed at site ν, but

u's account number is available locally. We use the notation if b $then$ s $else$ t as syntactic sugar for the term if-then-else(b, s, t), with the rewrite rules:

$$\text{if-then-else}(\mathsf{true}, X, Y) \to X$$
$$\text{if-then-else}(\mathsf{false}, X, Y) \to Y$$

The syntax used in this paper to associate sites to function definitions is just one of many alternative notations (another alternative is to use an object-oriented inspired syntax, writing $\nu.\mathsf{averagebalance}(u)$).

In this paper, we assume that the site where each function is defined is known and therefore the annotations used in function symbols are just constants. An interesting generalization consists of allowing the use of variables as annotations when the site is not known in advance, and considering the dynamic computation of site identifiers (for instance, a 'linker' program could dynamically generate the address of the site where averagebalance is defined).

4 *DEBAC* policy specifications via rewrite rules

In this section, we use an example to illustrate the use of distributed rewriting systems for specifying *DEBAC* policies. We do not claim that this is the only way to formalize a *DEBAC* policy as a rewrite system. Instead, our goal is to give an executable[1] specification of a *DEBAC* 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 unauthorized access is granted and no authorized access is denied).

4.1 Defining *DEBAC* policies

We assume that events are represented as ground terms of the form $\mathsf{event}(e_i, u, a, t)$, as discussed above. We specify a *DEBAC* policy by giving its *generic* and *specific* functions.

The generic functions are category and status, together with auxiliary functions such as user (see Section 3). We define these functions by the rules:

$$\begin{aligned}
\mathsf{category}(U, L) &\to F(\mathsf{status}(U, L))\\
\mathsf{status}(U, \mathsf{nil}) &\to \mathsf{cons}(c_0, \mathsf{nil})\\
\mathsf{status}(U, \mathsf{cons}(E, L)) &\to if\ U = \mathsf{user}(E)\\
&\ \ then\ \mathsf{cons}(\mathsf{Estatus}(E)), \mathsf{status}(U, L))\\
&\ \ else\ \mathsf{status}(U, L)
\end{aligned}$$

where c_0 is a default category (we could return the empty list instead), status looks for events involving a user U in the list L, and uses a *specific* function Estatus that associates a category c_i to a user according to the particular event

[1] For instance, the language MAUDE [15] can be used to execute rewrite-based specifications.

e_i in which the user was involved (this, of course, is specific to the application that we are modeling). The auxiliary function user extracts the user involved in a given event, as explained above. The function F, which is used in the definition of $\mathsf{category}$, takes as argument a list of categories associated to a user, according to the history of events, and returns one specific category. Again, the particular definition of F depends on the application. For example, we can take F to be the function head that returns the head of the list, or the functions max or min returning the highest or lowest category in the list, respectively; more elaborate functions are also possible.

Although we are focusing on first-order rewriting in this paper, the discussion above highlights an advantage of a higher-order rewriting formalism (such as, e.g., Combinatory Reduction Systems [24]). Indeed, in a higher-order rewriting system, $\mathsf{category}$ and status would be parameterized by F and $\mathsf{Estatus}$, respectively (i.e., they would be variables that can be instantiated with different functions).

Specific functions, as their name suggests, depend on the specific application that we are modeling. For example, in a bank scenario, we may define:

$$\begin{array}{l}\mathsf{Estatus}(\mathsf{event}(E, U, depositing, T))\\ \to if\ \mathsf{averagebalance}_\nu(\mathsf{account}(U)) > 10000\ and\ \mathsf{NotBlacklisted}_\mu(U)\\ then\ \mathsf{GOLD\text{-}CLIENT}\ else\ \mathsf{NORMAL\text{-}CLIENT}\end{array}$$

whereas in a university, students may acquire rights (change category) as they pass their exams:

$$\begin{array}{ll}\mathsf{Estatus}(\mathsf{event}(E, U, enroll, T)) & \to \text{REGISTERED-STUDENT}\\ \mathsf{Estatus}(\mathsf{event}(E, U, pay, T)) & \to \text{REGULAR}\\ \mathsf{Estatus}(\mathsf{event}(E, U, exams1styear, T)) & \to if\ \mathsf{pass}_\nu(U, 1styear)\\ & \quad and\ \mathsf{paid}_\mu(U, fees)\\ & \quad then\ \text{2ND-YEAR STUDENT}\\ & \quad else\ \text{IRREGULAR}\end{array}$$

where pass_ν and paid_μ are auxiliary functions returning boolean values.

Consider now the (chronologically ordered) list of events

$$\begin{array}{l}l = [\ \mathsf{event}(e_2, u, exams1styear, 20060130), \mathsf{event}(e_1, u, pay, 20060115),\\ \qquad \mathsf{event}(e_0, u, enroll, 20050901)]\end{array}$$

and assume that the function F, that is used in the definition of $\mathsf{category}$, is the function head returning the head of a list. Then we have

$$\begin{array}{l}\mathsf{category}(u, l) \to \mathsf{head}(\mathsf{status}(u, l)) \to^*\\ \mathsf{head}([\text{2ND-YEAR STUDENT, REGULAR, REGISTERED-STUDENT}])\end{array}$$

which finally leads to $\mathsf{category}(u, l) \to^*$ 2ND-YEAR STUDENT.

4.2 Evaluating access requests

Access requests from users can be evaluated by using a rewrite system to grant or deny the request according to the history of events and the user's category assignments that are specified in the *DEBAC* policy. For that, we may use the following rules, where a user u asks for an action a to be performed on a resource r accessible from a site μ. The symbols U, A, R, S, L are variables and the operator member is the standard membership test operator.

$$\begin{aligned} \mathsf{access}(A, U, R, S, L) &\rightarrow \mathsf{check}(\mathsf{member}((A, \mathsf{category}(U, L)), \mathsf{privileges}(R, S))) \\ \mathsf{check}(\mathsf{true}) &\rightarrow \mathsf{grant} \\ \mathsf{check}(\mathsf{false}) &\rightarrow \mathsf{deny} \end{aligned}$$

Here we assume that the function privileges returns a list of pairs (action, category allowed to perform that action) for a given resource in a given site. For example, privileges may be defined by rules such as:

$$\mathsf{privileges}(r, s) \rightarrow [(a_{11}, c_{11}), \ldots, (a_{1n}, c_{1n})]$$

In the discussion that follows, we will use R_{DEBAC} to refer to the rewrite system that contains the set of rules that we have defined so far.

4.3 Properties of the *DEBAC* policy

In order for a *DEBAC* 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_{DEBAC} are easy to check and will be used to show that the specification is consistent, correct and complete:

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

Proof. i) To prove termination, we use a modularity result for hierarchical unions (see Section 2 and [19]). First, observe that the system R_{DEBAC} is hierarchical: The auxiliary functions such as user, account, averagebalance, pass, paid, etc., form the basis of the hierarchy; they are terminating and non-duplicating. The specific rules defining Estatus, privileges and check form the first enrichment, and they are clearly terminating (they are not recursive). The second enrichment consists of the rules defining status and auxiliary functions, such as member. These are recursive functions, but the recursive calls are made on strict subterms of the arguments in the left-hand side of the rule. Finally, the non-recursive rules, defining category and access, complete the system.

We can therefore conclude that the system is terminating [19].

ii) 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 [27].

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

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

Property 2 (Consistency). For any list of events l, $u \in \mathcal{U}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, $s \in \mathcal{S}$: it is not possible to derive, from R_{DEBAC}, both grant and deny for a request $\mathsf{access}(a, u, r, s, l)$.

We can give a characterization of the normal forms:

Property 3. The normal form of a ground term of the form $\mathsf{access}(a, u, r, s, l)$ where $u \in \mathcal{U}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, $s \in \mathcal{S}$ and l is a list of events, is either grant or deny.

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

Property 4 (Totality). Each access request $\mathsf{access}(a, u, r, s, l)$ from a pre-authenticated user u to perform an action a on the resource r in a site s is either granted or denied.

Correctness and *Completeness* are also easy to check:

Property 5 (Correctness and Completeness). For any $u \in \mathcal{U}$, $a \in \mathcal{A}$, $r \in \mathcal{R}$, $s \in \mathcal{S}$ and list of events l:

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

Proof. Since the specification is consistent and total, it is sufficient to show that $\mathsf{access}(a, u, r, s, l) \rightarrow^* \mathsf{grant}$ if and only if u belongs to a category of users that is assigned the access privilege a on the resource r in s. This is easy to check in the examples above, by inspection of the rewrite rules.

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

The rewrite rules provide an executable specification of the policy (the rewrite rules are both a specification *and* an implementation of the access control function). The rules given above can be transformed into a MAUDE program by adding type declarations for the function symbols and variables used and by making minor syntactical changes.

5 Extensions of the *DEBAC* model

5.1 *DEBAC* with ordered categories

In RBAC models, it is usual to include role hierarchies to allow for the implicit specification of authorizations. This idea can be incorporated into the *DEBAC* model: we can accommodate a notion of a hierarchy of categories for users, where a user in category c_i will inherit, via a category hierarchy, the privileges of users in any category c_j such that $c_j < c_i$ with respect to a given partial ordering.

To represent a partial ordering on categories, we can add rules of the form

$$\mathsf{dpred}(c_i) \rightarrow [c_1, \ldots, c_j]$$

to specify a function $\mathsf{dpred} : \mathcal{C} \rightarrow \mathit{List}(\mathcal{C})$, where $\mathsf{dpred}(c_i) = [c_1, \ldots, c_j]$ means that $c_1, \ldots, c_j$ are direct predecessors of c_i in the ordering. Then we redefine the member function used in the definition of access (see Section 4.2), so that it takes into account the ordering (i.e., a category will have its privileges plus the privileges of all the categories that precede it in the ordering).

We use the following definition of member (where we omit the definition of the standard operations on sets and booleans):

$$\begin{aligned}
\mathsf{member}((A, C), \mathsf{nil}) &\rightarrow \mathsf{false} \\
\mathsf{member}((A, C), \mathsf{cons}((A', C'), L)) &\rightarrow \mathit{if} \quad A = A' \quad \mathit{and} \quad (C = C' \ \mathit{or} \ C' \in \mathsf{pred}(C)) \\
& \ \mathit{then} \ \ \mathsf{true} \\
& \ \mathit{else} \ \ \mathsf{member}((A, C), L)
\end{aligned}$$

$$\begin{aligned}
\mathsf{pred}(C) &\rightarrow \mathsf{dpred}(C) \cup \mathsf{preds}(\mathsf{dpred}(C)) \\
\mathsf{preds}(\mathsf{nil}) &\rightarrow \mathsf{nil} \\
\mathsf{preds}(\mathsf{cons}(C, L)) &\rightarrow \mathsf{pred}(C) \cup \mathsf{preds}(L)
\end{aligned}$$

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

5.2 *DEBAC* with constraints

A number of RBAC models with constraints, such as *separation of duties*, have been proposed. Constraints that are similar to separation of duties constraints can also be specified in a *DEBAC* model using rewrite rules, as an administrative check on a *DEBAC* policy.

Separation of duties is the property that specifies that categories assigned to a user cannot be mutually exclusive. To ensure that a specification of a *DEBAC* policy satisfies the separation of duties property, we will erase conflicting categories assigned to a user (producing a list of non-mutual-exclusive categories).

This is obtained by evaluation of $\mathsf{clean}(\mathsf{status}(u))$ in a rewrite system containing the rules:

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

6 Related work

In this section, we discuss our approach in relation to existing related literature. We note first that reduction systems have been used to model a variety of problems in security. For example, the SPI-calculus [1] was developed as an extension of the π-calculus for proving the correctness of authentication protocols. The π-calculus itself has been used to reason about a number of basic access control policies and access mechanisms (see, for example, [3]). Term rewriting has also been used in the analysis of security protocols [9], for defining policies for controlling information leakage [18], and for intrusion detection [2].

On the use of rewriting for access control specifications, the work by Koch et al [25] is related to our proposal. In [25], Koch et al describe a formalization of RBAC using a graph-based approach, with graph transformation rules used for describing the effects of actions as they relate to RBAC notions. More recently, [7, 29] use term rewrite rules to model access control policies. The formalization used by Koch et al provides a basis for proving properties of RBAC specifications, based on the categorical semantics of the graph transformations. The approach used in [7, 29] is operational: access control policies are specified as sets of rewrite rules and access requests are specified as terms which are evaluated using the rewrite rules. Our work addresses similar issues to [7, 25, 29] but provides a different formulation of access policies that is suitable for distributed environments. We also define a new type of access control model, *DEBAC*, that we argue generalizes RBAC. On the latter point, RBAC may be viewed as a special case of *DEBAC* in which the only necessary events are those involving security administrators performing the actions of user role assignment and user role deassignment, and the actions of permission assignment (to a role) and permission deassignment (from a role). The events of consequence that involve users are events of activating a role and deactivating a role. These events can be naturally accommodated in the *DEBAC* model; however, the *DEBAC* model additionally permits any number of other events to be represented.

The work on access control by Jajodia et al [20] and Barker and Stuckey [8] is also related to ours. In [20] and [8], access control requirements are represented in (constraint) logic programming languages. In these approaches, the requirements that must be satisfied in order for requesters to access resources are specified by using rules expressed in (C)LP languages and access request evaluation may be viewed as being performed by reducing an access request to a non-reducible clause (using, for example, SLG-resolution [31] or constraint solvers [26]). The term rewriting approach is similarly based on the idea of computation by reduction, and has similar attractions to the (C)LP approaches of Jajodia et al and Barker and Stuckey. However, in contrast to these approaches, our proposal does not require that the syntactic restriction to access policies that are *locally stratified* [6] be adopted (to ensure the existence of a categorical semantics and thus unambiguous access control policies). Moreover, we describe a form of access control model, the *DEBAC* model, that is applicable in the context of distributed access whereas [20] and [8] formally define access control models for centralized systems. The *DEBAC* model that we have described is also more expressive than any of the *Datalog*-based languages that have been proposed for distributed access control (see [5, 21, 17, 10]); these languages, being based on a monotonic semantics, are not especially well suited for representing dynamically *changing* distributed access request policies of the form that we have considered in this paper.

Work on temporal RBAC [8, 11], is related to our work on *DEBAC* in the sense that TRBAC models are concerned with the important notion of change and events. However, in [8] and [11], the events of interest are restricted to simple time/clock events. In the *DEBAC* model, any number of application-specific events (e.g., enrolling, passing_exams, etc.) may be represented, in addition to clock events. In [12], event expressions are used to trigger the enabling and disabling of roles. However, the proposal in [12] is based on an RBAC model that is quite different to the *DEBAC* model. Moreover, to ensure that a categorical semantics exists, syntactic restrictions are imposed on the language described in [12], to treat conflicting (prioritized) event expressions; such restrictions do not need to be imposed in our approach.

We also note that although our approach is based on rules and events the framework that we describe has a well defined declarative semantics that is quite different to the *ad hoc* operational semantics that are used in, for example, *active rule systems* [13]. Active rule systems are also based on the notions of rules and events but, unless some generally quite restrictive syntactic constraints are imposed, do not provide an adequate semantic basis for proving properties of access control policies (unlike term rewrite systems).

7 Conclusions and further work

We have described an event-based distributed access control model that we have developed to address certain shortcomings of RBAC models when the latter are applied in certain distributed computing contexts. Our *DEBAC* model takes the notion of an event as primitive (rather than a role), and has been designed to include features that specifically facilitate the autonomous changing of access control policy requirements, the proving of a wide range of properties of *DEBAC* policies (which follow directly from the syntax of *DEBAC* policy specifications), and the use of correct operational methods for the evaluation of user access requests with respect to distributed sources of access control and other forms of information (that do not need to be syntactically restricted in the way that other access policy specification languages are). For distributed access control, we introduced distributed term rewriting systems.

We note that the idea of access policy composition [30] is especially important in distributed environments (where access control information may need to be shared across multiple sites). Hence, a key matter for future work is to define appropriate algebras for *DEBAC* policy composition. A related matter for future work is to relax our assumption of atomic events and to treat access request evaluation in terms of sequences, disjunctions and conjunctions of events (for which an event algebra may be defined). We also intend to investigate the issue of evaluating access requests when conflicting information is received about the same user from different sites in a distributed system.

Acknowledgements

Part of this work was carried out at LIF, while the second author was visiting the MoVe team, partially supported by the Université de Provence.

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. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, Great Britain, 1998.
5. J. Bacon, K. Moody, and W. Yao. A model of OASIS RBAC and its support for active security. *TISSEC*, 5(4):492–540, 2002.
6. C. Baral and M. Gelfond. Logic programming and knowledge representation. *JLP*, 19/20:73–148, 1994.
7. S. Barker and M. Fernández. Term rewriting for access control. In *Proc. DB-Sec'2006*, volume 4127 of *LNCS*. Springer-Verlag, 2006.
8. 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.
9. 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 *LNCS*. Springer-Verlag, 2002.
10. M. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY 2004*, pages 159–168, 2004.
11. E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS*, 23(3):231–285, 1998.
12. E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. In *Proc. 5th ACM Workshop on Role-Based Access Control*, pages 21–30, 2000.

13. E. Bertino, B. Catania, and G. Zarri. *Intelligent Database Systems.* Addison Wesley, 2001.
14. P. Borovansky, C. Kirchner, H. Kirchner, and P-E. Moreau. ELAN from a rewriting logic point of view. *TCS*, 285:155–185, 2002.
15. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *RTA 2003*, volume 2706 in *LNCS*, pages 76–87. Springer-Verlag, 2003.
16. 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.
17. J. DeTreville. Binder, a logic-based security language. In *Proc. IEEE Symposium on Security and Privacy*, pages 105–113, 2002.
18. R. Echahed and F. Prost. Security policy in a declarative style. In *Proc. PPDP'05.* ACM Press, 2005.
19. M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Proc. ADT'94*, volume 906 in *LNCS*. Springer-Verlag, 1995.
20. S. Jajodia, P. Samarati, M. Sapino, and V.S. Subrahmaninan. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.
21. T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symp. Security and Privacy*, pages 106–115, 2001.
22. C. Kirchner, H. Kirchner, and M. Vittek. *ELAN user manual.* Nancy (France), 1995. Technical Report 95-R-342, CRIN.
23. 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.
24. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *TCS*, 121:279–308, 1993.
25. M. Koch, L. Mancini, and F. Parisi-Presicce. A graph based formalism for rbac. In *Proc. SACMAT'04*, pages 129–187, 2004.
26. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.
27. M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.
28. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
29. A. Santana de Oliveira. Rewriting-based access control policies. In *Proc. of SECRET'06*, ENTCS. Elsevier, 2007.
30. D. Wijesekera and S. Jajodia. Policy algebras for access control the predicate case. In *ACM Conf. on Computer and Communications Security*, pages 171–180, 2002.
31. *The XSB System Version 2.7.1, Programmer's Manual*, 2005.