A Framework for the Analysis of Access Control Policies with Emergency Management

Sandra Alves[1]

Dept. of Computer Science, University of Porto, Porto, Portugal

Maribel Fernández[2,3]

Dept. of Informatics, King's College London, London WC2R 2LS, UK

Abstract

We define a framework for the analysis of access control policies that aims at easing the specification and verification tasks for security administrators. We consider policies in the category-based access control model, which has been shown to subsume many of the most well known access control models (e.g., MAC, DAC, RBAC). Using a graphical representation of category-based policies, we show how answers to usual administrator queries can be automatically computed, and properties of access control policies can be checked. We show applications in the context of emergency situations, where our framework can be used to analyse the interaction between access control and emergency management.

Keywords: Security Policies, Access Control, Operational Semantics, Rewriting, Graph-based analysis.

1 Introduction

Access control systems are used to protect resources against unauthorised use. In its most basic form, an access control policy specifies the actions that each user is allowed to perform on each resource. A pair of a resource and an action is called a *permission*.

A variety of access control models and languages for access control policy specification are currently in use. One of the most popular is the ANSI (hierarchical) role-based access control (H-RBAC) model [2], where users are assigned to roles and each role is assigned a set of permissions (extensions of RBAC, using time and location constraints, are discussed in e.g., [19]). More flexible models, such as the event-based access control (DEBAC) model [12] and the action-status access control

[1] Email: sandra@dcc.fc.up.pt

[2] Email: maribel.fernandez@kcl.ac.uk

[3] Partially funded by the European Office of Aerospace Research and Development (EOARD-AFOSR).

model [7], specify permissions that depend on dynamic conditions, defined in terms of events that happen in the system.

A metamodel for access control, which can be specialised for domain-specific applications, has been proposed in [5]. It identifies a core set of principles of access control, abstracting away many of the complexities that are found in specific access control models, in order to simplify the tasks of policy writing and policy analysis. A key aspect of the metamodel is to focus attention on the notion of a *category*. A category is a class of entities which share some property. Classic types of groupings used in access control, like a role, a security clearance, a discrete measure of trust, etc., are particular instances of the more general notion of category. In category-based access control policies, permissions are assigned to categories of users (which we refer to as principals), rather than to individual users. Categories can be defined on the basis of e.g., user attributes, geographical constraints, resource attributes. For example, users may be assigned to different categories according to their age, and a policy can give a permission to perform an action (e.g., download) on a resource (e.g., a film) to users in the category "older than 12" but not in the category "child". In this way, permissions change in a dynamic and autonomous way (e.g., when a registered user has a birthday), unlike, e.g., role-based access control models, which require the intervention of a security administrator.

Given the complexities and scope involved in the definition of access control policies, formal methods to analyse and reason about access control policies are essential. This is particularly important in the case of systems dealing with access control in the context of emergency situations, where users' rights may need to change in order to cope with specific emergencies. Formal specifications of access control models and policies (see, for instance, [16,30]) have used theorem provers, purpose-built logics, and, more recently, functional and rewriting-based approaches (see, for example, [29,12]). A rewrite-based operational semantics for the category-based metamodel was described in [14], where the expressive power of the metamodel is also demonstrated. Using standard rewriting tools, rewrite-based security policies can be verified to ensure that each access request has a unique answer, as shown in [15].

In this paper, we define a framework for the analysis of access control policies that aims at easing the specification and verification tasks for security administrators. We consider category-based policies, since the category-based model subsumes the most well known access control models [14], thus allowing us to obtain a generic framework. Using a graphical representation of policies, we show how answers to usual administrator queries can be automatically computed, and properties of access control policies (such as, every access request receives a unique answer) can be checked. We show applications of the framework to the analysis of policies in distributed environments, and in particular policies that include management of rights in emergency situations. For example, in a hospital environment, an access control policy may specify that each doctor has access to the patient records of his/her own patients. However, if a patient p has a cardiac arrest, then any doctor in the ward should have access to p's medical records. We will show that this kind of policies can be easily specified in our framework, in a visual and formal way, and properties, such as the fact that the policy ensures a "separation of duty" constraint (where no user should be allowed to perform two conflicting actions on the same resource), can be easily proved using graph-based algorithms and rewriting techniques.

Overview of the paper.

The remainder of the paper is organised as follows. In Section 2, we recall the category-based access control model. Section 3 discusses emergency policies. Section 4 presents a graph-based framework to represent and analyse category-based policies, and Section 5 describes its implementation (a Ruby application for policy visualisation and analysis). In Section 6, we discuss related work, and in Section 7, conclusions are drawn and further work is suggested.

2 Preliminaries: The Category-Based Metamodel

We assume familiarity with basic notions on first-order logic and term-rewriting systems [3]. We briefly describe below the key concepts underlying the category-based metamodel of access control; see [5] for a detailed description.

Informally, a category is any of several distinct classes or groups to which entities may be assigned. Entities are denoted by constants in a many sorted domain of discourse, including: a countable set $\mathcal{C}$ of categories, denoted $c_0, c_1, \ldots$; a countable set $\mathcal{P}$ of principals, denoted $p_0, p_1, \ldots$ (we assume that principals that request access to resources are pre-authenticated); a countable set $\mathcal{A}$ of named *actions*, denoted $a_0, a_1, \ldots$; a countable set $\mathcal{R}$ of *resource identifiers*, denoted $r_0, r_1, \ldots$; a finite set $\mathcal{A}uth$ of possible *answers* to access requests (e.g., {grant, deny, undetermined}) and a countable set $\mathcal{S}$ of *situational identifiers* to denote environmental information. More generally, entities are represented by terms (e.g., a principal is represented by a data structure $principal(p_i, attributeList)$), but constants will be sufficient for most examples in this paper.

The metamodel includes the following relations:

- *Principal-category assignment:* $\mathcal{PCA} \subseteq \mathcal{P} \times \mathcal{C}$, such that $(p, c) \in \mathcal{PCA}$ iff a principal $p \in \mathcal{P}$ is assigned to the category $c \in \mathcal{C}$.
- *Permission-category assignment:* $\mathcal{ARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$, such that $(a, r, c) \in \mathcal{ARCA}$ iff the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ can be performed by principals assigned to the category $c \in \mathcal{C}$.
- *Authorisations:* $\mathcal{PAR} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{R}$, such that $(p, a, r) \in \mathcal{PAR}$ iff a principal $p \in \mathcal{P}$ can perform the action $a \in \mathcal{A}$ on the resource $r \in \mathcal{R}$.

Definition 2.1 [Axioms] The relation $\mathcal{PAR}$ satisfies the following core axiom, where we assume that there exists a relationship $\subseteq$ between categories; this can simply be equality, set inclusion (the set of principals assigned to $c \in \mathcal{C}$ is a subset of the set of principals assigned to $c' \in \mathcal{C}$), or a specific relation may be used.

$(a1)$ $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},$

$$\exists c, c' \in \mathcal{C}, ((p, c) \in \mathcal{PCA} \wedge\ c \subseteq c'\ \wedge (a, r, c') \in \mathcal{ARCA}) \Leftrightarrow (p, a, r) \in \mathcal{PAR}$$

Definition 2.2 [Category-based policy] A category based policy is a tuple $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR} \rangle$, where $\mathcal{E} = (\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{S})$, such that axiom $(a1)$ is satisfied.

Operationally, axiom $(a1)$ can be realised through a set of functions, as shown in [14]. We recall the definition of the function $\mathsf{par}(P, A, R)$ below; it relies on functions pca, which returns the list of categories assigned to a principal, and arca, which returns a list of permissions assigned to a category.

Definition 2.3 A rewrite-based specification of the axiom $(a1)$ in Def. 2.1 is given by the rewrite rule:

$$(a2)\ \mathsf{par}(P, A, R) \to if\ \ (A, R) \in \mathsf{arca}^*(\mathsf{contain}(\mathsf{pca}(P)))\ then\ \mathsf{grant}\ else\ \mathsf{deny}$$

As the function name suggests, $\mathsf{contain}$ computes the set of categories that contain any of the categories given in the list $\mathsf{pca}(P)$. The function $\in$ is a membership operator on lists, grant and deny are answers, and arca^* generalises the function arca to take into account lists of categories:

$$\mathsf{arca}^*(\mathsf{nil}) \to \mathsf{nil} \qquad \mathsf{arca}^*(\mathsf{cons}(C, L)) \to \mathsf{append}(\mathsf{arca}(C), \mathsf{arca}^*(L))$$

An access request by a principal p to perform the action a on the resource r can then be evaluated simply by rewriting the term $\mathsf{par}(p, a, r)$ to normal form.

The axiom $(a1)$, and its algebraic version $(a2)$, state that a request by a principal p to perform the action a on a resource r is authorised only if p belongs to a category c such that for some category below c (e.g., c itself) the action a is authorised on r, otherwise the request is denied. There are other alternatives, e.g., considering *undeterminate* as answer if there is not enough information to grant the request.

An axiomatisation of *distributed category-based access control* was proposed in [11] to specify federative policies, obtained as a composition of individual access control policies. In a federation, each member has its own access control policy, and contributes to the definition of a global access control policy. We will use this notion of distributed access control to define emergency policies in the next section. We recall the main axioms below.

Assume the set $\mathcal{S}$ of situational identifiers includes identifiers for sites, i.e., $s \in \mathcal{S}$ identifies one of the components of the federation. $\mathcal{PCA}_s$, $\mathcal{ARCA}_s$, and $\mathcal{PAR}_s$ denote families of relations indexed by site identifiers. Intuitively, $\mathcal{PAR}_s$ denotes the authorisations that are valid in the site s. The relation $\mathcal{PAR}$ defining the global authorisation policy is obtained by composing the local policies defined by the relations $\mathcal{PAR}_s$ as indicated below. The sets $\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}$ include, respectively, the principals, categories, actions and resources in any of the sites of the system, which are assumed to be globally known in the federation (alternatively we can define sets $\mathcal{P}_s$, $\mathcal{C}_s$, $\mathcal{A}_s$, $\mathcal{R}_s$ for each site).

Definition 2.4 [Distributed Axioms] The *distributed category-based metamodel* is defined by the following core axioms

$(b1)$ $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},\ \forall s \in \mathcal{S},$

$$(\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA}_s \wedge c \subseteq c'\ \wedge (a, r, c') \in \mathcal{ARCA}_s) \Leftrightarrow (p, a, r) \in \mathcal{PAR}_s$$

$(f1)$ $\forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},$

$$(p, a, r) \in \mathcal{OP}_{par}(\{\mathcal{PAR}_s \mid s \in S\}) \Leftrightarrow (p, a, r) \in \mathcal{PAR}$$

The result of an access request may be different depending on the site where the request is evaluated. The axiom $(f1)$ describes the global authorisation relation, which is obtained from the ones defined at each site by using the operator $\mathcal{OP}_{par}$. While most of the existing policy languages (e.g., XACML) have a fixed set of operators to combine policies, the metamodel can accommodate a large range of composition operators.

Definition 2.5 A distributed category-based policy is defined by the tuple $\langle \mathcal{E}, \{\mathcal{PCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{ARCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{PAR}_i\}_{i\in\mathcal{S}}, \mathcal{OP}_{par}\rangle$, such that axioms $(b1)$ and $(f1)$ are satisfied.

The operational semantics of the distributed model is defined by extending the functions presented in Definition 2.3, using distributed term rewrite systems (DTRSs), which are term rewrite systems where rules are partitioned into modules, each associated with a unique identifier, and function symbols are annotated with such identifiers (for more details on DTRSs, we refer to [12]). In other words, specific functions defined in a particular site are indexed by the site identifier; functions with no site annotations are assumed to be defined locally.

Definition 2.6 In a distributed environment, the rewrite-based specification of the axiom $(b1)$ in Def. 2.4 is given by the rewrite rule:

$$(b2)\ \mathsf{par}_s(P, A, R) \rightarrow if\ (A, R) \in \mathsf{arca}^*_s(\mathsf{contain}(\mathsf{pca}_s(P)))\ then\ \mathsf{grant}\ else\ \mathsf{deny}$$

where the function $\in$ is a membership operator on lists, grant and deny are answers, and arca^*_s is the function defining the assignment of privileges to categories, as in the previous section.

The axiom $(f1)$ can be realised by the following rewrite rule, which implements $\mathcal{OP}_{par}$ through the use of par_s, where the function fauth combines the results into a final answer according to the operator op:

$$(f2)\ \mathsf{authorised}(p, a, r, s_1, \ldots, s_n) \rightarrow \mathsf{fauth}(\mathsf{op}, \mathsf{par}_{s_1}(p, a, r), \ldots, \mathsf{par}_{s_n}(p, a, r)).$$

The axiom $(f1)$ can be implemented in several ways. The version chosen in the definition above corresponds to a very general rewrite rule that can be used for evaluating an access request in a single central site (if $n = 1$ and the operator op is the identity), as well as for evaluating combinations of answers (with a suitable operator op) from n different local policies. Functions such as $\mathsf{psite}(p)$, which returns the site where the principal p is registered, or $\mathsf{rsite}(r)$ which returns the site where the resource r is located, may be used. In this way, access requests can be evaluated in a predefined central site, or priority can be given to local evaluation, or more elaborated combinations of access answers can be implemented. We refer to [13,10] for examples.

3 Emergency policies

In this paper we consider a particular kind of policy composition, where an access control policy is combined with an emergency policy that specifies how various emergency situations affect the rights of users to access resources. In this approach, events are elementary or compound actions [22], which we represent using terms of the form $\mathsf{event}(e_i, p, a, o, t, l)$, following [14]. Here, event is a data constructor, e_i is an event identifier, p is a principal associated to the event, a is an action, o its object, t is the time when the event happened, and l is a list of arguments (depending on the event type, some arguments might not be required). Emergency policies will be associated to specific events. To simplify, we consider only atomic events, and assume that a history of all events that happened in the system is available (e.g., via a log). We follow the definition of emergency given in [18]:

An emergency takes place at time T if an event E happened at a time T_s which is earlier than T, and resulted in the initiation of the emergency, and this emergency has not been ended before T as a consequence either of (i) clipping, i.e., an event E' happening at a time T' between T_s and T that causes the emergency to be terminated or (ii) expiring a timeout δ for this emergency.

For example, in a hospital environment, an access control policy may specify that each doctor has access to the patient records of his/her own patients. However, if a patient p has a cardiac arrest, then any doctor in the ward should have access to p's medical records during the cardiac emergency.

The distributed metamodel and the notion of event defined above can be used to specify access control in emergency situations. We consider two sites π_1 and π_2 such that π_1 contains a standard policy and π_2 contains an emergency policy. In the previous example, let *patient* be a category consisting of all patients (of a given hospital), and *doctor* be a category consisting of all doctors (of the given hospital). Let $doctor(X)$ be a (parameterised) category consisting of all doctors of the patient X, such that for all X, $doctor(X) \subseteq doctor$, i.e., the category $doctor(X)$ inherits all permissions from the category *doctor*. Assume the relations $\mathcal{PCA}$ and $\mathcal{ARCA}$ satisfy the following axioms, where $emerg(bcrd, P)$ is true if an event initiating a cardiac emergency for P has been detected, and no event ending the emergency has been recorded:

$$\forall P,\ (P, patient) \in \mathcal{PCA}\ \Rightarrow\ (read, record(P), doctor(P)) \in \mathcal{ARCA}_{\pi_1}$$

$$\forall P,\ (P, patient) \in \mathcal{PCA}\ \wedge\ emerg(bcrd, P) \Rightarrow\ (read, record(P), doctor) \in \mathcal{ARCA}_{\pi_2}$$

Operationally, we specify rewrite rules for arca in the standard (π_1) and emergency (π_2) sites, and combine the policies using a union operator with priority to grant.

$$\begin{aligned}
\mathsf{arca}_{\pi_1}(doctor(P)) &\rightarrow [(read, record(P))] \\
\mathsf{arca}_{\pi_2}(doctor) &\rightarrow [(read, record(P)) \mid P \in patientList \wedge emerg(bcrd, P)])
\end{aligned}$$

where *patientList* returns the list of patients, that is, P such that $patient \in \mathsf{pca}(P)$. We discuss in the following section techniques to prove properties of such policies, e.g., to show that any doctor has access to the record of a patient suffering a cardiac emergency.

4 Analysis of Category-Based Policies

4.1 Graph Representation of Policies

We start by defining how policies can be represented by means of graphs. Graphical or visual representations of data structures and algorithms have a number of established and significant advantages over textual representations. In particular, they tend to be easier to understand and analyse than the corresponding textual representations. Furthermore, being a well-studied area, algorithms and properties of graph theory can be used to analyse properties of policies.

Definition 4.1 [Policy graph] We define a *policy graph*, or *graph* for short, as a tuple $\mathcal{G} = (\mathcal{V}, E, lv, le)$, where $\mathcal{V}$ is a set of nodes, E is a set of undirected edges, which is a subset of $\{\{v_1, v_2\} \mid v_1, v_2 \in \mathcal{V} \wedge v_1 \neq v_2\}$, lv is an injective labelling function mapping nodes to entities in the category-based metamodel $lv : \mathcal{V} \to \mathcal{P} \cup \mathcal{C} \cup \mathcal{A} \cup \mathcal{R}$, and le is a labelling function for edges.

We assume the usual notion of *degree* of a node, as the number of edges connected to that node.

Definition 4.2 A *path* in $\mathcal{G}$ of length n, between two nodes v_0, v_n, is a sequence $v_0, v_1, \ldots, v_n$, such that $\{v_{i-1}, v_i\} \in E$ for all $1 \leq i \leq n$.

Since we have different types of nodes, we define a function $\mathsf{type} : \mathcal{V} \to \{P, C, A, R\}$, which associates each node with the type of its label. More precisely, $\mathsf{type}(v) = P$ if $lv(v) = p \in \mathcal{P}$ (that is, P is the type of the nodes representing principals), and, similarly, C is the type of nodes representing categories, A actions, and R resources. Furthermore, we consider the type of an edge to be determined by the type of the nodes connected by that edge, that is, we consider a function $\mathsf{type} : E \to \{P, C, A, R\} \times \{P, C, A, R\}$. An edge-type will be a pair (T_1, T_2), which for simplicity we will represent as T_1T_2. For example, AC is the type of an edge connecting a node of type A with a node of type C. Note that, since our edges are undirected, we do not distinguish between the types T_1T_2 and T_2T_1.

We will use types to restrict the edges of graphs representing policies, as follows.

Definition 4.3 [Well-typed policy graph] A policy graph is well typed if it contains only the following kinds of edges

(a) $\{v_1, v_2\} \in E$ such that $\mathsf{type}(v_1) = P \wedge \mathsf{type}(v_2) = C$, which connect principals to categories. This corresponds to an edge of type PC.

(b) $\{v_1, v_2\} \in E$ such that $\mathsf{type}(v_1) = C \wedge \mathsf{type}(v_2) = A$, which connect categories to actions. This corresponds to an edge of type CA.

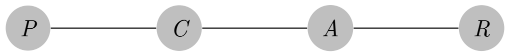
(c) $\{v_1, v_2\} \in E$ such that $\mathsf{type}(v_1) = A \wedge \mathsf{type}(v_2) = R$, which connect actions to resources. This corresponds to an edge of type AR.

Definition 4.4 [Relations associated with $\mathcal{G}$] Let $\mathcal{G}$ be a well-typed policy graph and let $\mathcal{PC}$, $\mathcal{CA}$ and $\mathcal{AR}$ be the set of all edges of types PC, CA and AR, respectively. Then we define the following relations associated with $\mathcal{G}$:

- The relation $\mathcal{PCA}_{\mathcal{G}}$ is defined by the set $\{(lv(v_1), lv(v_2)) \mid \mathsf{type}(v_1) = P \wedge \{v_1, v_2\} \in \mathcal{PC}\}$. It can also be defined by the set of paths of size 1, starting from nodes of type P.

- The relation $\mathcal{ARCA}_\mathcal{G}$ is given by the set $\{(lv(v_1), lv(v_2), lv(v_3)) \mid \mathsf{type}(v_1) = A \wedge \{v_1, v_2\} \in \mathcal{AR} \wedge \{v_3, v_2\} \in \mathcal{CR}\}$. It can also be defined by the set of paths of size 2, starting from nodes of type C and ending on nodes of type R (or vice-versa).
- The relation $\mathcal{PAR}_\mathcal{G}$ is given by the set $\{(lv(v_1), lv(v_3), lv(v_4)) \mid \exists v_2 \; s.t. \; \{v_1, v_2\} \in \mathcal{PC} \wedge \{v_2, v_3\} \in \mathcal{CA} \wedge \{v_3, v_4\} \in \mathcal{AR}\}$. It can also be defined by the set of paths of size 3, starting from nodes of type P and ending on nodes of type R (or vice-versa).

Proposition 4.5 *Any path of size 3 in a policy graph, starting in a node of type P and ending in a node of type R, must have the following shape:*



Proof. Direct consequence of the restriction imposed on edge types. Note that all the paths of length 1 starting in a node of type P end on a node of type C. The paths of length 1 starting from a node of type C end on a node of type A or a node of type P and the paths of length 1 starting from a node of type A end on a node of type C or a node of type R. Hence, the only paths of size 3 starting in a node of type P and ending in a node of type R are the ones that traverse a node of type C and a node of type A. □

Therefore, from all the paths of size 3 starting in a node of type P and ending in a node of type R, one can effectively compute the $\mathcal{PAR}$ relation (for the moment we are not taking into account the $\subseteq$ relation between categories). Furthermore, it is easy to see that any well-typed policy graph represents an access control policy.

Proposition 4.6 *Each well-typed policy graph $\mathcal{G}$ defines a unique category-based access control policy $\langle \mathcal{E}, \mathcal{PCA}_\mathcal{G}, \mathcal{ARCA}_\mathcal{G}, \mathcal{PAR}_\mathcal{G} \rangle$.*

Proof. According to Def. 2.2, we need to prove that $\mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA}_\mathcal{G}$ and $\mathcal{PAR}_\mathcal{G}$ satisfy axiom $(a1)$, which follows by Def. 4.4 (for now we are considering that the relation $\subseteq$ between categories is the equality relation, but we will deal with the general case later). □

However, for a given policy, there may be more than one graph that generates the policy; for example, take any graph that differs on the unassigned permissions (that is, differing on edges between actions and resources such that there is no edge connecting the action to any category). There is, however, a unique minimal graph corresponding to the policy.

Definition 4.7 [Types for paths] Let $v_0, v_1, \ldots, v_n$ be a path of length n, such that $\mathsf{type}(v_i) = T_i$ for $0 \leq i \leq n$. The type of the path is the sequence given by the types of the edges along the path, that is $T_0T_1, T_1T_2, \ldots, T_{n-1}T_n$.

The relations $\mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA}_\mathcal{G}$ and $\mathcal{PAR}_\mathcal{G}$ can now be defined in terms of typed-paths of a certain type. For example:

$$\mathcal{PAR}_\mathcal{G} = \{(lv(v_1), lv(v_3), lv(v_4)) \mid v_1, v_2, v_3, v_4 \text{ is a path of type } PC, CA, AR\}.$$

Note also that one can define the minimum unique graph as the set of paths of type CA and the set of paths of type CA, AR.

This formalisation of the category-based metamodel as graphs does not take into account distributed policies. This can be obtained by considering that for each site there is a special additional node in the graph, with type S, labelled with the site identifier, and to which all the principals of that site are connected.

Definition 4.8 [Distributed policy graph] Let $\mathcal{G} = (\mathcal{V}, E, lv, le)$ be a well-typed policy graph representing a policy and let $s \in \mathcal{S}$ be a location identifier in the distributed system. Then $\mathcal{G}_s$, the policy graph of site s, is defined by $(\mathcal{V} \cup \{v_s\}, E \cup \{\{v_s, v\} \mid v \in \mathcal{V} \wedge \mathsf{type}(v) = P\}, lv', le')$, where lv' and le' extend lv and le in the natural way, that is, by mapping v_s to s.

A distributed policy graph is a tuple of graphs $(\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n})$ where $\mathcal{G}_{s_i} = (\mathcal{V}_i, E_i, lv'_i, le'_i)$ is the policy graph for s_i, $1 \leq i \leq n$. For each location $s \in \mathcal{S}$, the relations $\mathcal{PCA}_s$, $\mathcal{ARCA}_s$ and $\mathcal{PAR}_s$ are defined (as paths on $\mathcal{G}_s$) as in the non-distributed scenario.

Using graphs to formalise the global authorisation policy $\mathcal{PAR}$ is not so straightforward. One possibility to define $\mathcal{PAR}$ is simply by the union, but there are much more sophisticated ways to combine the policies. For now we will take the union of the different policies.

Definition 4.9 [Union graph] Let $(\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n})$ be a distributed policy graph with n sites $s_1, \ldots, s_n$, where each $\mathcal{G}_{s_i} = (\mathcal{V}_i, E_i, lv_i, le_i)$. Their associated *union-graph* is defined by $\mathcal{G} = (\mathcal{V}, E, lv, le)$, where $\mathcal{V}$ is a set of nodes such that for each $v_i \in \mathcal{V}_i$ where $lv_i(v_i) = x$ there exists a unique node $\nu \in \mathcal{V}$ such that $lv(\nu) = x$, E is a multiset of edges such that if $\{v_1, v_2\} \in E_i$, then there is an edge $\{\nu_1, \nu_2\} \in E$ where $lv_i(v_1) = lv(\nu_1)$ and $lv_i(v_2) = lv(\nu_2)$ and $le_i(\{v_1, v_2\}) = le(\{\nu_1, \nu_2\})$.

Since multiple edges can connect the same pair of nodes, this corresponds to a multigraph. Note that the entities of the metamodel (principals, categories, actions and resources) can be known by different sites. This means that in the graph representation of the global policy one has to be able to distinguish whether a node/edge belongs to a particular site or not. To that end, we define two functions $\mathsf{visible}_{\mathcal{V}} : \mathcal{V} \to 2^{\mathcal{S}}$ and $\mathsf{visible}_E : E \to \mathcal{S}$, such that, for each node v, $\mathsf{visible}_{\mathcal{V}}(v)$ will return the set of sites that have a node with the same label as v, that is, the set of sites where v is known, and similarly for each edge e $\mathsf{visible}_E(e)$ will return e's site.

Definition 4.10 A *path* of length n in the union-graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, v_1, \ldots, v_n$, such that $\{v_{i-1}, v_i\} \in E$ for all $1 \leq i \leq n$, and all the edges $e_1, \ldots, e_n$ in the path are visible in the same site.

Proposition 4.11 *A union graph $\mathcal{G}$, defines a distributed category-based policy where $\mathcal{OP}_{par}$ is a union operator with priority to grant.*

Proof. According to Def. 2.5, we need to prove that axioms $(b1)$ and $(f1)$ are satisfied. Axiom $(b1)$ follows from each $\mathcal{G}_{s_i}$, and for axiom $(f1)$ if there exists a path of type PC, CA, AR ($\mathsf{visible}_E$ is always the same for all the edges in the path), then this path also exists in some $\mathcal{G}_{s_j}$, which means that $\mathsf{par}(p, a, r) \in \mathcal{PAR}_{s_j}$, therefore it will belong to $\mathcal{OP}_{par}\{\mathcal{PAR}_s \mid s \in \mathcal{S}\}$ if $\mathcal{OP}_{par}$ is a union operator with priority to grant. $\square$

One could consider copies of the same entity as different nodes, which could be linked, and then collapsed into a single node when visually displaying the graph. We will discuss later, different options for effectively displaying a policy. The $\mathsf{visible}_{\mathcal{V}}, \mathsf{visible}_E$ functions defined above will be useful in the definition of visualisation algorithms. In terms of graph representation, other more sophisticated combinations of policies could be defined taking into account, for example, conflicting information, but we will deal with that in the future.

Up to this point we have not taken into account in our graph formalisation of the metamodel the $\subseteq$ relation between categories. As mentioned before, this could be achieved by defining a function $\mathsf{contain}$ on nodes of type C. However, this would no longer allow us to define certain properties based solely on paths.

Alternatively we formalise the $\subseteq$ relation in terms of edges of type CC. That is, we allow the set of edges to contain edges of the form $\{c_1, c_2\}$, such that $\mathsf{type}(c_1) = \mathsf{type}(c_2) = C$. Note that edges are undirected, however, in $\subseteq$ one might have $c_1 \subseteq c_2$ but not $c_2 \subseteq c_1$. Therefore, when defining paths involving edges of type CC one needs to know in which direction these edges can be transversed. We define a function target on edges such that, if $v_i \in \mathsf{target}(\{v_1, v_2\})$ then v_i can be a destination node of that edge (note that both v_1, v_2 can be destination nodes of an edge, if it can be transversed in both directions). We need a more constrained notion of path taking into account the target function, so we will refine Definition 4.10:

Definition 4.12 A *constrained path* of length n in $\mathcal{G} = (\mathcal{V}, E, lv, le)$, is a sequence $v_0, v_1, \ldots, v_n$, such that, for all $1 \leq i \leq n$, $\{v_{i-1}, v_i\} \in E \wedge v_i \in \mathsf{target}(\{v_{i-1}, v_i\})$.

Definition 4.13 Let c_1, c_2 be two categories in $\mathcal{C}$, then $c_1 \subseteq c_2$ if there is a constrained path of type $(CC)^*$[4] between the nodes labelled by c_1 and c_2.

In this paper we are only considering a relation $\subseteq$ between categories, but the same could be considered for other entities (for example resources or actions).

Definition 4.14 [Dynamic policy graph] A *dynamic policy graph* is a well-typed graph $\mathcal{G} = (\mathcal{V}, E, lv, le)$ together with a function ld on $\mathcal{V}$ such that $ld(v) = R$, for some convergent rewrite system R satisfying the following conditions:

- if $\mathsf{type}(v) = P$, then $ld(v)$ defines a function pca, which returns (for each p) a list of categories;
- if $\mathsf{type}(v) = C$, then $ld(v)$ defines a function arca, which (for each c) returns a list of permissions (pairs of the form *(action,resource)*).

Relations between the entities in our model can change in an autonomous way (e.g. due to events that happen in the system), with principals/permissions being added or removed from certain categories. In this sense a policy graph can be seen as a photo-shot of the system at a particular time. From the graph one can extract

[4] We consider the usual notation of a^* to refer to a sequence of the form $\underbrace{a, a, \ldots, a}_{n}$, with $n \geq 0$.

the relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{PAR}$ at a particular instant, but not how to get the next photo. This is the purpose of the ld function.

Definition 4.15 A dynamic policy graph is said to be correct (at a particular instant) iff:

- for every node v of type P, $ld(v) = [c_1, \ldots, c_n]$ iff there exists an edge $\{v, v_i\}$ in E for $i = 1, \ldots, n$, such that $lv(v_i) = c_i$;
- for every node v of type C, $ld(v) = [(a_1, r_1), \ldots, (a_n, r_n)]$ iff there exists in E edges $\{va_i, vr_i\}$ and $\{v, va_i\}$ for $i = 1, \ldots, n$ such that $lv(va_i) = a_i$, $lv(vr_i) = r_i$ and $lv(v) = c$;

A dynamic policy graph represents a dynamic category-based policy. Each request has a unique answer if the associated dynamic graph is correct.

4.2 Analysis of static properties based on graphs

For a given policy, we are interested in checking certain properties in terms of principals, categories, resources, and permissions. We first consider a non-distributed system. As examples of static properties one can be interested in checking, we consider the following: (i) Are all the principals associated with at least one category? (ii) Are there permissions associated to all categories? (iii) Are all the resources in effective use (in terms of principals and permissions)? (iv) For a given category, who are the associated principals? (v) To which categories belongs a given principal? (vi) For a given category, what are the associated permissions? (vii) For a given principal, what are the associated permissions?

For a given policy represented by a well-typed graph $\mathcal{G} = (\mathcal{V}, E, lv, le)$, the properties above can be formalised in the following way:

(i) All the principals are associated with at least one category if the degree of every node of type P is positive (in a distributed scenario where all the principals are connected to the site(s) node(s), then this property is guaranteed by ensuring that all the principals are connected to a node of type C);

(ii) For each node of type C there is a path of type $(CC)^*, CA, AR$.

(iii) For each node of type R there is a path of type $PC, (CC)^*, CA, AR$.

(iv) For a given node of type C, find all the neighbours of type P.

(v) For a given node of type P, find all the neighbours of type C.

(vi) For a given node of type C, find all the paths of type $(CC)^*, CA, AR$. The last two nodes of each path will define a permission associated to that category.

(vii) For a given node of type P, find all the paths of type $PC, (CC)^*, CA, AR$. The last two nodes will define the permissions associated to that principal.

Proposition 4.16 *All the checks above can be computed in polynomial time with respect to* $|V| + |E|$.

Note that the properties mentioned above are still valid in a distributed scenario, either by considering paths in the individual graphs $\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n}$, or the notion of paths in the union-graph defined above.

Other static properties can be checked using properties on the underlying graph of the policy, specifically for the distributed scenario. For example, detecting whether there are permissions that are in conflict. If an action a_1 in a resource r is in conflict with an action a_2 in r, then for every principal p there should only be one path of type $PC, (CC)^*, CA, AR$ linking p and r in the union graph.

One can also ask, for a given union graph, what is the minimum unique graph that corresponds to a policy. This can be computed by starting in nodes of type S and considering the edges that are in the spanning tree considering only branches that end in nodes of type P, C and R. This way one eliminates edges between nodes of type A and R that are not associated to any category, and edges between nodes of type C and R that are not associated to any resource.

There are more complex and relevant questions that can be dealt with using this formalism. For example:

- For a given policy-graph $\mathcal{G}$, and given sets of principals $\{p_1, \ldots, p_n\}$ and permissions $\{(a_1, r_1), \ldots, (a_n, r_n)\}$, what is a minimum number of changes (in term of adding/deleting elements in the $\mathcal{PCA}$ and $\mathcal{ARCA}$ relations) necessary to ensure that, collectively, the given set of principals has the given permissions?

  Although this is a generally complex problem, by fixing either the number of principals or permissions we can obtain manageable instances of the problem.
- For a given policy-graph $\mathcal{G}$, and given sets of principals $\{p_1, \ldots, p_n\}$ and permissions $\{(a_1, r_1), \ldots, (a_n, r_n)\}$, what is the minimal number of edges necessary to guarantee that those, and those alone, permissions are available to those users? One might be interested in guaranteeing a particular set of permission assignment, but in a controlled manner, particularly in a emergency scenario. For a single pair $(p, (a, r))$, this corresponds to finding the shortest path in the graph, connecting the principal to the permission.

4.3 Application: Emergency Management

Our main motivation is to provide an analysis framework to deal with policy updates, allowing security administrators to detect changes introduced into a policy in a scenario involving emergency situations. As mentioned in Section 3, an emergency policy can be modelled using an additional emergency site. The graph representation of a policy in an emergency scenario will be given by combining the graphs of the normal policy with the emergency policy (taking an appropriate composition operator). For example, when combining the two policies using a union operator with priority to grant, then the union-graph defined above will suffice.

The analysis described in the last section can then be used to specify properties when dealing with the emergency. For example, one guarantees that in the case of a patient i suffering a cardiac emergency any doctor has access to his/her medical record, by showing that, for every principal p in the category *doctor*, there exists a path of type $PC, (CC)^*, CA, AR$ of the form $p, c^+, access, record_i$ in the graph associated to the emergency policy.

The graphs of the normal and emergency policies can be used to analyse other properties. For example, one might wish to determine what are the permissions revoked by the emergency; the permissions created by the emergency; whether or not every principal has a certain permission during the emergency; whether or not a certain action is forbidden during the emergency, etc. All these properties can easily be established using our graph formalisation. "Separation of duties" constraints also correspond to path constraints. For example, the constraint *"no user has permission to both activate an alarm (triggering an emergency and possibly acquiring more permissions) and delete the emergency log (which records which users have activated alarms)"* holds if the set of paths of type $PC, (CC)^*, CA, AR$ in the policy graph does not include a path ending in $activate, alarm$ and a path ending in $delete, log$, and starting in the same node of type P.

5 A tool to analyse policies

In this section we describe a tool to analyse policies using the graph formalisation.

5.1 Visual representation of policies

As we mentioned before, using graphs to formally represent policies in the category based metamodel has numerous advantages in terms of algorithms and well established properties from graph-theory that can be used to analyse properties of these policies, as well as tools for displaying graphs in an effective way.

Recall that we defined a distributed system as a tuple $(\mathcal{G}_1, \ldots, \mathcal{G}_n)$, and the union-graph $\mathcal{G}$ as the multigraph combining all the information from the individual policies $\mathcal{G}_i$. Also recall that, each $\mathcal{G}_i$ contains a special node of type S, to which every principal is connected. Since entities in $\mathcal{P} \cup \mathcal{C} \cup \mathcal{A} \cup \mathcal{R}$ can be global to all the sites an appropriate visual display of the system should clearly show in which sites each entity is. Ideally one would like to represent the system in a 3-dimensional space, where each site would be at a different horizontal plane, where the identities common to several sites would be vertically aligned. Different horizontal planes could be selected and merged to combine the policies. Given the degree of complexity in manipulating/displaying this type of graphics, we consider alternative options:

- All the different $\mathcal{G}_i$ are represented in the same plane, but different colours are used to identify the different sites (if a node is known in several sites then it is surrounded by a different colour ring for each site it belongs to).
- By selecting a node site (a node of type S), then all the colours of the other sites turn to grey, therefore highlighting the selected site.
- Since we are representing a multigraph, there can be several edges between the same nodes. They should be represented using the colour associated to their site.
- To further highlight a selection, the size of the involved nodes can be augmented (or the nodes not involved can be diminished).

Note that, unconnected entities cannot be associated to any site, in which case the function $\mathsf{visible}_{\mathcal{V}}$ can be used to determine the colour(s) of the nodes. When representing the union-graph, the function $\mathsf{visible}_E$ is also used to determine the colour of the edges. Other possibilities could be considered such as representing different copies of the same entity corresponding to the different sites where it occurs, and using especial edges to connect the different copies. Each policy could then be displayed separately, with the connection to other policies being given by the additional edges. This alternative would give a clear visualisation of each individual policy (particularly if the additional edges could be omitted), but would not provide a proper overview of the general policy.

5.2 A Ruby prototype

An application called *Policy Manager* [28] was implemented to provide an easy to use graphical tool for security administrators, allowing the construction and management of multiple policies. The application was implemented in Ruby [21]: an interpreted, object-oriented, multi-paradigm programming language.

In terms of graphical display of policy data, the Policy Manager provides user-friendly visual representations that facilitate the task of identifying policy flaws. The application provides the user with a complete view of a policy as a tree, allowing users to zoom in on overcrowded sections of the tree. It also allows the selection of particular entities, highlighting the nodes and edges associated to that entity. For example, by clicking on a principal name, the tree will centre on the selected object, allowing a clearer view of the categories and permissions associated to that principal. Furthermore, the user is able to reposition the elements by dragging, as well as remove irrelevant elements from the view. The application also comprises a textual view, allowing for simple queries concerning the policy.

One key aspect of the project was the implementation of the dynamic behaviour of categories. Unlike roles in RBAC, categories can change dynamically based on events or changes in the state of the system (emergencies can be seen as specific kinds of events). To represent dynamic graphs (see Def. 4.14), the tool allows the user to save Ruby code describing events in the database. This, however, requires the users to have knowledge of the Ruby language, as well as raising security issues. A more desirable solution, would be to define a user-friendly Domain Specific Language, to allow users to specify categories and permissions, for example, using rewrite rules. This language could then be compiled into code to be inserted in the policy database (as is currently done with the Ruby code used to specify categories).

6 Related Work

Several formal languages have been used in the literature to model and analyse access control problems. Koch et al. [26] use graphs to formalise RBAC, in particular by modelling role management operations by graph transformation rules. More recently, [6,29,12] use term rewrite rules to model particular access control models and to express access control policies. Our approach combines the use of a graph formalism to represent a concrete state of the system, and the use of rewrite rules to model the dynamics of the system. The Generalised TRBAC model [25] and ASL [23] aim at providing a general framework for the definition of policies, however they focus essentially on the notion of users, groups and roles (interpreted as being synonymous with the notion of job function). Li et al.'s *RT* family of role-trust models [27] provides a general framework specialised for defining specific policy requirements (in terms of credentials).

The specification of policies by means of rewriting systems allows, not only to take advantage of the extensive theory of rewriting to establish security properties, as shown in [29,17,10] amongst other works, but also to make use of rewriting-based frameworks (such as CiME, MAUDE or TOM) to reason about policy properties. Our work addresses similar issues, but is based on a notion of category-based access control for distributed environments, which we interpret using labelled graphs, and which can be instantiated to include concepts like times, events, and histories that are not included as elements of *RT* or RBAC. In [15], CiME is integrated in a tool designed to automatically check consistency and totality of RBAC access control policies. A similar technique could be used to analyse the rewrite system in a dynamic policy graph.

The framework that we have described is more expressive than any of the Datalog-based languages that have been proposed for distributed access control (see [4,24,20,8]); these languages, being based on a monotonic semantics, are not especially well suited for representing dynamically changing distributed policies. Another work dealing with decentralised systems is reported in [9], where the authors propose the constraint logic programming language SecPal for specifying a wide range of authorisation policies and credentials, using predicates defined by clauses. In our approach, we focus on graph interpretations of a general metamodel suitable for distributed systems rather than on the design of a specification language, but the operational semantics of the metamodel could serve as a basis for a policy definition language.

7 Conclusions and Further Work

This paper describes a framework that aims at aiding the specification and analysis of access control policies, by using a graph-based formalism to represent policies and relying on graph properties to extract policy properties. In Section 4 we focus on properties that are mostly static, but we are also interested in other (dynamic) properties (such as verifying that at any point in time, each access request to a resource by a principal will always receive a unique answer), which are related to the operational semantics (defined using term rewriting). In future work, to analyse dynamic properties of policies and help administrators develop and manage policy updates, we plan to develop a version of Policy Manager within PORGY [1], a tool that allows users to visualise and simulate systems via port-graph rewriting.

Additionally, in the context of an analysis application such as Policy Manager, one would be interested in being able to describe dynamic behaviour using a user-friendly Domain Specific Language, suitable for policy administrators. We believe that a rewriting-based language could be an appropriate solution to that problem. This would provide an implementation of the operational semantics of the category-based metamodel in Policy Manager, could be integrated with tools such as CiME to verify desirable properties, and translated into other programming languages for integration in policy analysis tools.

Acknowledgements:

We thank Anatoli Degtyarev for many valuable discussions on the topics of this paper, and Hossein Mirzapour-Aghdaghi for implementing the Policy Manager tool.

References

[1] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. Porgy: Strategy-driven interactive transformation of graphs. In *TERMGRAPH*, volume 48 of *EPTCS*, pages 54–68, 2011.

[2] ANSI. RBAC, 2004. INCITS 359-2004.

[3] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, Great Britain, 1998.

[4] J. Bacon, K. Moody, and W. Yao. A model of OASIS RBAC and its support for active security. *TISSEC*, 5(4):492–540, 2002.

[5] S. Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the ACM Int. Conf. SACMAT 2009*, pages 187–196. ACM Press, 2009.

[6] S. Barker and M. Fernández. Term rewriting for access control. In *Data and Applications Security. Proceedings of DBSec'2006*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[7] Steve Barker. Action-status access control. In Volkmar Lotz and Bhavani M. Thuraisingham, editors, *SACMAT*, pages 195–204. ACM, 2007.

[8] M. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY 2004*, pages 159–168, 2004.

[9] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proc. of CSF 2007*, pages 3–15. IEEE Computer Society, 2007.

[10] C. Bertolissi and M. Fernández. A rewriting framework for the composition of access control policies. In *Proceedings of PPDP'08*. ACM Press, 2008.

[11] C. Bertolissi and M. Fernández. Rewrite specifications of access control policies in distributed environments. In *Proc. of STM 2010*, number 6710 in LNCS. Springer, 2011.

[12] C. Bertolissi, M. Fernández, and S. Barker. Dynamic event-based access control as term rewriting. In *Data and Applications Security XXI. Proc. of DBSEC 2007*, volume 4602 of *LNCS*. Springer, 2007.

[13] C. Bertolissi and Maribel Fernández. Distributed event-based access control. *International Journal of Information and Computer Security, Special Issue: selected papers from Crisis 2008*, 3(3–4), 2009.

[14] C. Bertolissi and Maribel Fernández. Category-based authorisation models: operational semantics and expressive power. In *Proceedings of ESSOS'10*, LNCS. Springer, 2010.

[15] Clara Bertolissi and Worachet Uttha. Automated analysis of rule-based access control policies. In *Proc. of PLPV*, pages 47–56. ACM, 2013.

[16] P. A. Bonatti and P. Samarati. Logics for authorization and security. In *Logics for Emerging Applications of Databases*, pages 277–323. Springer, 2003.

[17] Tony Bourdier, Horatiu Cirstea, Mathieu Jaume, and Hélène Kirchner. Formal specification and validation of security policies. In *FPS*, pages 148–163, 2011.

[18] Barbara Carminati, Elena Ferrari, and Michele Guglielmi. A system for timely and controlled information sharing in emergency situations. *IEEE Trans. Dep. Sec. Comput.*, 10(3):129–142, 2013.

[19] S. M. Chandran and J. B. D. Joshi. Lot-rbac: A location and time-based rbac model. In *Proc. of WISE'05*, volume 3806 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2005.

[20] J. DeTreville. Binder, a logic-based security language. In *Proc. IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[21] D. Flanagan and Y. Matsumoto. *The Ruby programming language - everything you need to know: covers Ruby 1.8 and 1.9.* O'Reilly, 2008.

[22] Michael Gelfond and Jorge Lobo. Authorization and obligation policies in dynamic systems. In *ICLP*, pages 22–36, 2008.

[23] S. Jajodia, P. Samarati, M. Sapino, and V.S. Subrahmaninan. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.

[24] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symp. Security and Privacy*, pages 106–115, 2001.

[25] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. Knowl. Data Eng.*, 17(1):4–23, 2005.

[26] M. Koch, L. Mancini, and F. Parisi-Presicce. A graph based formalism for RBAC. In *SACMAT*, pages 129–187, 2004.

[27] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[28] H. Mirzapour-Aghdaghi and M. Fernández. Policy Manager: a tool to analyse category-based access control policies, 2014. Final Year Project, King's College London, UK. http://policymanager.herokuapp.com

[29] A. Santana de Oliveira. *Rcriture et Modularit pour les Politiques de Scurit.* PhD thesis, Université Henri Poincare, Nancy, France, 2008.

[30] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, and Martin Gogolla. Analyzing and managing role-based access control policies. *IEEE Trans. Knowl. Data Eng.*, 20(7):924–939, 2008.