A Graph-Based Framework for the Analysis of Access Control Policies

Sandra Alves[1a], Maribel Fernández[2b]

[a]*Dept. of Computer Science, University of Porto, Porto, Portugal*
[b]*Dept. of Informatics, King's College London, London WC2R 2LS, UK*

Abstract

We design a graph-based 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 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, 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 (RBAC) model [5], where users, which we refer to as principals, are assigned to roles and each role is assigned a set of permissions (extensions using time and location constraints are discussed in, e.g., [25]). More flexible models, such as the event-based access control (DEBAC) model [20] and the action-status access control model [11], specify permissions that depend on dynamic conditions, defined in terms of

[1]This work is partially funded by the ERDF through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT as part of project UID/EEA/50014/2013.
[2]Work partially funded by EOARD - project FA8655-10-1-3047.

events that happen in the system. Extensions to RBAC to take into account context-based constraints are defined in e.g., [40].

A metamodel for access control, which can be specialised for domain-specific applications, has been proposed in [9]. 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 that 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 (CBAC) policies, permissions are assigned to categories of users, rather than to individual users. Categories can be defined on the basis of e.g., user attributes, geographical constraints, resource attributes, etc. For example, 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 15" 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 to update role permissions.

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 [14]. 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, [22, 50]) have used theorem provers, purpose-built logics, and, more recently, functional and rewriting-based approaches (see, for example, [49, 20]). Using standard rewriting tools, rewrite-based policies can be verified to ensure, for example, that each access request has a unique answer [19, 37, 21]. A rewrite-based operational semantics for CBAC policies is described in [17], where their expressive power is also demonstrated. Distributed CBAC policies are defined in [19].

In this paper, we define a graph-based framework for the analysis of access control policies that aims at easing the specification and verification tasks for security administrators. Graphical or visual representations of data structures and algorithms have significant advantages over textual representations, as they tend to be easier to understand and analyse. Furthermore, being a well-studied area, algorithms and properties of graph theory can be used to analyse properties of policies. We consider category-based policies, since the category-based model subsumes the most well known access control models [17, 19], thus allowing us to obtain a generic framework. In addition to authorisations, the policies we consider may also specify prohibitions. Using a graphical representation of policies, we show how answers to usual administrator queries (such as, is every resource covered by the policy?) 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 dis-

tributed (multi-site) environments, where the global system policy is defined as a composition of individual policies that apply at each site. In addition, we consider 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 only to the medical 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. This kind of policy can be seen as a composition of a standard policy and an emergency policy, and can be specified in our framework in a visual and formal way. Properties, such as a "separation of duty" constraint, where no user should be allowed to perform two conflicting actions on the same resource (e.g., issue a purchase order and approve it), can be easily checked 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 presents a graph-based framework to represent category-based policies, which we use in Section 4 to analyse such policies. In Section 5 we give an application in the context of emergency policies. Section 6 describes implementations of this formalism. In Section 7, we discuss related work, and in Section 8, conclusions are drawn and further work is suggested.

This paper is a revised and extended version of [3]. The extensions include dealing with policies that specify prohibitions as well as authorisations, and with distributed (multi-site) policies obtained as a composition of local site policies. In addition, here we consider the graph representation of dynamic policies, where the authorisation and prohibition relations evolve depending on events that take place in the system (without needing the intervention of the security administrator), and we show how graph-based policies can be used to answer typical security administrator queries.

2. Preliminaries: Category-Based Access Control Policies

We assume familiarity with basic notions on first-order logic and term-rewriting systems [6]. We briefly describe below the key concepts underlying the category-based metamodel of access control, following [19] (see also [9]).

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, ...; a countable set $\mathcal{P}$ of principals, denoted p_0, p_1, ...; a countable set $\mathcal{A}$ of named *actions*, denoted a_0, a_1, ...; a countable set $\mathcal{R}$ of *resource identifiers*, denoted r_0, r_1, ... and a countable set $\mathcal{S}$ of *situational identifiers* to denote environmental information.

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 $r \in \mathcal{R}$.

These relations can be defined extensionally (by enumeration of the tuples), in which case we say that the policy is *static*, or by comprehension, using relevant predicates, which might take into account the system state, in which case we say that the policy is *dynamic*.

Definition 1 (Axioms) The relation $\mathcal{PAR}$ satisfies the following core axiom, where we assume that there exists a relationship $\subseteq$ between categories (a partial ordering, which can simply be equality or set inclusion, i.e., $c \subseteq c'$ if 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).

$$(a1) \quad \forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},\\ (\exists c, c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c \subseteq c' \wedge (a, r, c') \in \mathcal{ARCA}) \Leftrightarrow (p, a, r) \in \mathcal{PAR}$$

Definition 2 (CBAC policy) A category-based access control (CBAC) policy is a tuple $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR} \rangle$, where $\mathcal{E} = (\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{S}, \subseteq)$, such that axiom $(a1)$ is satisfied.

Operationally, axiom $(a1)$ can be realised through a set of functions, as shown in [17]. We recall the definition of the function par below; it relies on functions pca, which returns the list of categories assigned to a principal, and arca, which returns the list of permissions assigned to a category.

Definition 3 A rewrite-based specification of axiom $(a1)$ in Def. 1 is given by the rewrite rule:

$$(a1') \quad \mathsf{par}(P, A, R) \quad \rightarrow \quad if\ (A, R) \in \mathsf{arca}^*(\mathsf{contain}(\mathsf{pca}(P)))\ \ then\ \mathsf{grant}\ \ else\ \mathsf{deny}$$

The function $\mathsf{contain}$ 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:

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

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 $(a1')$, 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 following the well-known *negative closed world* assumption [14]. However, many systems permit the definition of positive and negative authorisations (or authorisations and prohibitions).

CBAC policies can handle prohibitions as well as authorisations, using the relations $\mathcal{BARCA}$ and $\mathcal{BAR}$:

- Banned actions on resources: $\mathcal{BARCA} \subseteq \mathcal{A} \times \mathcal{R} \times \mathcal{C}$, such that $(a, r, c) \in \mathcal{BARCA}$ iff the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ is forbidden for principals assigned to the category $c \in \mathcal{C}$.
- Banned access: $\mathcal{BAR} \in \mathcal{P} \times \mathcal{A} \times \mathcal{R}$, such that $(p, a, r) \in \mathcal{BAR}$ iff performing the action $a \in \mathcal{A}$ on the resource $r \in \mathcal{R}$ is forbidden for the principal $p \in \mathcal{P}$.

A relation $\mathcal{UNDET}$ can also be defined if $\mathcal{PAR}$ and $\mathcal{BAR}$ are not complete, i.e., if there are access requests that are neither authorised nor denied (thus producing an undetermined answer).

If the relation $\mathcal{BARCA}$ is admitted, the following axioms should be satisfied:

$(a2)$ $\forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}$
$(\exists c \in \mathcal{C}, \exists c' \in \mathcal{C}, (p, c) \in \mathcal{PCA} \wedge c' \subseteq c \wedge (a, r, c') \in \mathcal{BARCA}) \Leftrightarrow$
$(p, a, r) \in \mathcal{BAR}$

$(a3)$ $\forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}$
$((p, a, r) \notin \mathcal{PAR} \wedge (p, a, r) \notin \mathcal{BAR}) \Leftrightarrow (p, a, r) \in \mathcal{UNDET}$

$(a4)$ $\mathcal{PAR} \cap \mathcal{BAR} = \varnothing$

Note that $(a2)$ states $c' \subseteq c$, whereas in $(a1)$ it is the reverse. Hence, a category inherits permissions from junior categories, and prohibitions from senior ones.

Definition 4 (CBAC policy with prohibitions) A CBAC policy with prohibitions is a tuple $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \mathcal{BARCA}, \mathcal{BAR}, \mathcal{UNDET} \rangle$, where $\mathcal{E} = (\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{S}, \subseteq)$, such that axioms $(a1)$-$(a4)$ are satisfied.

Below the term CBAC policy will be used to refer to category-based access control policies in general (with or without prohibitions).

A distinctive feature of CBAC policies is the fact that all the relations may evolve in time (without the intervention of the security administrator), as long as the axioms are satisfied. In other words, the allocation of principals to categories and the allocation of rights and prohibitions to categories may depend on the system state. In this sense, CBAC policies are *dynamic*.

An axiomatisation of *distributed category-based access control* was proposed in [18] to specify federative policies 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 Section 5. We recall the main axioms of the distributed category-based model 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. $\mathcal{PCA}_s$, $\mathcal{ARCA}_s$,

$\mathcal{BARCA}_s$, $\mathcal{UNDET}_s$, $\mathcal{PAR}_s$ and $\mathcal{BAR}_s$, where $s \in \mathcal{S}$, denote families of relations indexed by site identifiers. Intuitively, $\mathcal{PAR}_s$ (respectively $\mathcal{BAR}_s$) denotes the authorisations (resp. prohibitions) 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 $\mathcal{PAR}_s$ (using operators $\mathcal{OP}_{par}$ and $\mathcal{OP}_{bar}$ that are specific to the global policy). 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 5 (Axioms for distributed CBAC) The *distributed category-based model* is defined by the following core axioms.

$$
\begin{array}{ll}
(b1) & \forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R},\ \forall s \in \mathcal{S}, \\
& (\exists c, c' \in \mathcal{C}, (p,c) \in \mathcal{PCA}_s \wedge c \subseteq c'\ \wedge (a,r,c') \in \mathcal{ARCA}_s) \Leftrightarrow \\
& \qquad\qquad (p,a,r) \in \mathcal{PAR}_s \\
(c1) & \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, \forall s \in S, \\
& (\exists c, c' \in \mathcal{C}, (p,c) \in \mathcal{PCA}_s \wedge c' \subseteq c \wedge (a,r,c') \in \mathcal{BARCA}_s) \Leftrightarrow \\
& \qquad\qquad (p,a,r) \in \mathcal{BAR}_s \\
(d1) & \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, \forall s \in \mathcal{S}, \\
& ((p,a,r) \notin \mathcal{PAR}_s \wedge (p,a,r) \notin \mathcal{BAR}_s) \Leftrightarrow (p,a,r) \in \mathcal{UNDET}_s \\
(e1) & \forall s \in \mathcal{S}, \mathcal{PAR}_s \cap \mathcal{BAR}_s = \varnothing \\
(f1) & \forall p \in \mathcal{P},\ \forall a \in \mathcal{A},\ \forall r \in \mathcal{R}, \\
& (p,a,r) \in \mathcal{OP}_{par}(\{\mathcal{PAR}_s, \mathcal{BAR}_s \mid s \in S\}) \Leftrightarrow (p,a,r) \in \mathcal{PAR} \\
(g1) & \forall p \in \mathcal{P}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}, \\
& (p,a,r) \in \mathcal{OP}_{bar}(\{\mathcal{PAR}_s, \mathcal{BAR}_s \mid s \in \mathcal{S}\}) \Leftrightarrow (p,a,r) \in \mathcal{BAR} \\
(h1) & \mathcal{PAR} \cap \mathcal{BAR} = \varnothing
\end{array}
$$

The axioms $(b1)$-$(e1)$ generalise $(a1)$-$(a4)$. According to axioms $(b1)$ and $(c1)$, the result of an access request may be different depending on the site where the request is evaluated, since each site s has its own authorisation policy defined by the local relations $\mathcal{PAR}_s$ and $\mathcal{BAR}_s$. Note that $(p,a,r) \in \mathcal{UNDET}_s$ if and only if the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ is neither allowed nor forbidden for the principal $p \in \mathcal{P}$ at site $s \in \mathcal{S}$; hence, every tuple in $\mathcal{P} \times \mathcal{A} \times \mathcal{R}$ is in $\mathcal{PAR}_s \cup \mathcal{BAR}_s \cup \mathcal{UNDET}_s$. The axioms $(e1)$ and $(h1)$ preclude inconsistent specifications (i.e., a request cannot be both authorised and forbidden). The global authorisations and prohibitions (axioms $(f1)$ and $(g1)$) are obtained by composing the local relations, using the operators $\mathcal{OP}_{par}$ and $\mathcal{OP}_{bar}$, respectively, which are application-dependent. For example, in some applications a request should be denied if any of the component policies denies it (i.e., a "deny takes precedence principle" [34] applies), whereas in other cases, grant takes precedence. A "first-applicable" principle takes a list of policies and returns the answer corresponding to the first policy that produces grant or deny (it returns undetermined only if no policy in the list returns grant or deny). We illustrate below the definition of a deny-takes-precedence operator in CBAC.

Example 6 Consider a system with two sites $s, t \in \mathcal{S}$. To define a global policy where deny takes precedence, it is sufficient to use the operators $\mathcal{OP}_{bar} = (\mathcal{BAR}_s \cup \mathcal{BAR}_t)$ and $\mathcal{OP}_{par} = ((\mathcal{PAR}_s / \mathcal{BAR}_t) \cup (\mathcal{PAR}_t / \mathcal{BAR}_s))$. This corresponds to a union operator giving priority to deny, since tuples in $\mathcal{BAR}_s \cup \mathcal{BAR}_t$ are removed from the authorisation relation by the operator $\mathcal{OP}_{par}$. The definition of a union operator with priority to grant in CBAC is also straightforward.

While most of the existing policy languages (e.g., XACML [47]) have a fixed set of operators to combine policies, the axioms $(f1), (g1)$ allow us to accommodate a large range of composition operators. We refer to [16, 15] for examples.

Definition 7 (Distributed CBAC policy) A distributed category-based access control policy is defined by a tuple:

$$\langle \mathcal{E}, \{\mathcal{PCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{ARCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{BARCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{PAR}_i\}_{i\in\mathcal{S}}, \{\mathcal{BAR}_i\}_{i\in\mathcal{S}}, \{\mathcal{UNDET}_i\}_{i\in\mathcal{S}}, \mathcal{OP}_{par}, \mathcal{OP}_{bar} \rangle,$$

where $\mathcal{E} = (\mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{S}, \subseteq)$, such that axioms $(b1)$-$(h1)$ are satisfied.

We refer to [19] for a rewrite-based operational semantics of the distributed model, defined by extending the functions presented in Definition 3.

3. Graph Representation of Policies

Access control policies specify the authorisations and prohibitions that should be enforced in any implementation of the system. Policies should therefore include the authorisations that are needed for users to be able to carry out their tasks, but no more. To facilitate the task of specifying policies and checking that they are correctly defined, it is essential to design policy languages with a precise semantics and easy to use by security administrators (who are in charge of the creation and update of policies). Small changes in policy rules can have enormous consequences, which means that not only should policy languages be easy to use but also easy to analyse. For this reason, in this paper we advocate the use of formal languages that offer *visual* representations of policies. More precisely, we propose a *graph-based* language to represent policies.

In this section, we start by defining how static CBAC policies are represented by means of graphs (we consider first policies with only positive authorisations and then policies with both positive and negative authorisations, that is, authorisations and prohibitions). We then extend the framework in order to deal with distributed policies. Finally we show how to incorporate dynamic features, obtaining a graph formalism that is sufficiently expressive to represent the full CBAC metamodel.

To represent CBAC policies we use *labelled undirected graphs*, where nodes and edges represent entities and relations in the model, respectively.

We choose undirected graphs instead of directed graphs because we are representing relations rather than functions, and there is no preferred orientation

for edges: we use policies not only to evaluate requests but also to check properties, for which we need to traverse edges in both directions. For instance, if we represent the fact that p is in the category c by an edge between the nodes representing p and c, then we may need to go from p to c when p issues an access request (i.e., we need to check that p's category permits this access) but we may also need to traverse the edge from c to p to check properties of c (for example, to check whether the set of principals in c is non-empty; see Section 4.1, Q4).

Labels are attached to nodes and edges, and store data (in the form of pairs attribute-value) of relevance for the policy. More precisely, labels are records, as defined below.

Notation 8 (Record) Let a_i range over a finite set of attribute names, and t_i range over a finite set of values. A record $\mathsf{R} \in \mathcal{REC}$, is a term of the form $\{a_1 = t_1, \ldots, a_n = t_n\}$. We use the notation $\mathsf{R}.a$ for attribute selection, and the notation $\mathsf{update}(\mathsf{R}, a, t)$ to modify the value of attribute a in record R to t; if the attribute a does not exist in R, i.e. $\mathsf{R}.a$ is undefined, then the field a is added, with value t.

In our representation of policies below, we assume all records have an attribute ent with the name of the entity to which the record belongs.

Example 9 The record containing information about the principal John Lewis could be specified as follows: $\mathsf{R} = \{\mathsf{ent} = \mathit{JohnLewis}, \mathsf{type} = P, \mathsf{birthdate} = 19931216\}$, then $\mathsf{R.ent}$ is equal to *JohnLewis*, and $\mathsf{update}(\mathsf{R}, \mathsf{qualification}, \mathit{intern})$ results in the record $\{\mathsf{ent} = \mathit{JohnLewis}, \mathsf{type} = P, \mathsf{birthdate} = 19931216, \mathsf{qualification} = \mathit{intern}\}$.

3.1. Representing static CBAC policies

We consider first CBAC policies with positive authorisations only, satisfying axiom $(a1)$.

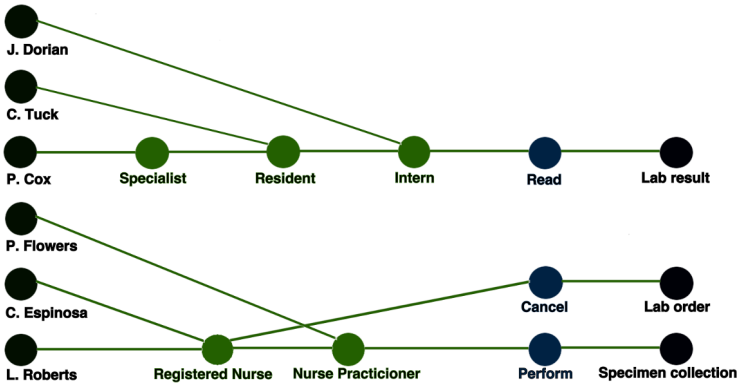
Definition 10 (Policy graph) A *policy graph*, or *graph* for short, is a tuple $\mathcal{G} = (\mathcal{V}, E, lv, le)$, where $\mathcal{V}$ is a set of nodes, E is a set of undirected edges, $lv : \mathcal{V} \to \mathcal{REC}$ is a labelling function for nodes, such that, for every $v \in \mathcal{V}$, $lv(v).\mathsf{ent} \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{A} \cup \mathcal{R}$, and $le : E \to \mathcal{REC}$ is a labelling function for edges, such that, for every $e \in E$ between nodes v_1 and v_2, $le(e).\mathsf{adj} = \{v_1, v_2\}$, where $v_1, v_2 \in \mathcal{V}$ and $v_1 \neq v_2$. In addition, we assume that the record labels of nodes contain a field $\mathsf{type} = T$, where $T \in \{P, C, A_{\mathcal{R}}, R\}$, such that $lv(v).\mathsf{type} = P$ if $lv(v).\mathsf{ent} = p \in \mathcal{P}$ (that is, P is the type of the nodes representing principals), and, similarly, C is the type of nodes representing categories and R resources. Types of the form $A_{\mathcal{R}}$ are used to type nodes representing actions. These types are indexed with a set $\mathcal{R} = \{r_1, \ldots, r_n\}$, representing the nodes of type R to which the action can apply. In general, unless it is needed, we will omit this index and write only A when referring to types of action nodes. The type of an edge is determined by the type of its adjacent nodes, that is, if $le(e).\mathsf{adj} = \{v_1, v_2\}$, then $\mathsf{type}(e) = (lv(v_1).\mathsf{type}, lv(v_2).\mathsf{type})$.

For simplicity we will represent edge-types (T_1, T_2) 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. Since our edges are undirected, we do not distinguish between the types T_1T_2 and T_2T_1.

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

Definition 11 A *path* of length n in a policy graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, v_1, \ldots, v_n$ of pairwise distinct nodes, such that, for all $1 \leq i \leq n$, $\{v_{i-1}, v_i\} = le(e).\mathsf{adj}$, for some $e \in E$.

Example 12 Consider a hospital where principals are categorised as patients, registered nurses, nurse practitioners, interns, resident doctors and specialists. The following diagram represents a policy graph, showing for each node the value of the ent attribute. Nodes of different types are shown in different colours. For example, node J.Dorian has type P, nodes Specialist, Resident and Intern have type C, node Read has type A, and node Labresult has type R.



In a policy graph, nodes should be uniquely identified by the fields ent and type. We will use the notation $v_1 \equiv v_2$ if $lv(v_1)$ and $lv(v_2)$ have the same values for ent and type. Furthermore, we will use types to restrict the edges of graphs representing policies (see Definition 18 below).

Our goal is to be able to compute the $\mathcal{PAR}$ relation of a CBAC policy directly from the graph representation. If there is no $\subseteq$ relation between categories, from all the paths of length 3 starting in a node of type P and ending in a node of type R, one can effectively compute the $\mathcal{PAR}$ relation. In order to deal with hierarchical category definitions and be able to compute the $\mathcal{PAR}$ relation based on paths, we formalise the $\subseteq$ relation in terms of edges of type CC (that is, edges between nodes $\{v_1, v_2\}$, such that $lv(v_1).\mathsf{type} = lv(v_2).\mathsf{type} = C$). Note that edges are undirected, however, in $\subseteq$ one might have $c_1 \subseteq c_2$ but not $c_2 \subseteq c_1$. Thus, we define an attribute target on edges of type CC to specify the destination node of the edge (note that both extremities could be destination nodes, if the nodes represent equivalent categories).

Since the relation between categories is hierarchical, we need a notion of path taking into account the target attribute:

Definition 13 A *constrained path* of length n in a policy graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, e_1, v_1, e_2 \ldots, e_n, v_n$, where all the nodes are different, such that for all $1 \leq i \leq n$, $le(e_i).\mathsf{adj} = \{v_{i-1}, v_i\} \wedge v_i \in le(e_i).\mathsf{target}$, if target exists in $le(e_i)$.

Definition 14 Let $\mathcal{G}$ be a policy graph and c_1, c_2 be two categories in $\mathcal{C}$. We write $c_1 \subseteq c_2$ if there is a constrained path between the nodes labelled by c_1 and c_2 where all the edges are of type (CC). If $c_1 \subseteq c_2$ and $c_2 \subseteq c_1$ then the categories c_1, c_2 are equivalent and we write $c_1 = c_2$.

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 15 (Types for paths) Let $v_0, \ldots, v_n$ be a path of length n, such that $lv(v_i).\mathsf{type} = T_i$ 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 notation $\mathsf{type}(v_0, v_1, \ldots, v_n) = T_0T_1, T_1T_2, \ldots, T_{n-1}T_n$ will be used to indicate that there is a path $v_0, v_1, \ldots, v_n$ and its edges have types $T_0T_1, T_1T_2, \ldots, T_{n-1}T_n$.

Furthermore, if an edge e between nodes v_i and v_{i+1} in a path $v_0, v_1, \ldots, v_n$ has type CC, then we will denote its type as $\overrightarrow{CC}$ if $v_{i+1} \in le(e).\mathsf{target}$, and $\overleftarrow{CC}$ if $v_i \in le(e).\mathsf{target}$ (that is, type $\overrightarrow{CC}$ means that the edge is traversed from a category c_i to a category c_{i+1} where $c_i \subseteq c_{i+1}$, that is, from a senior category to a junior category, and type $\overleftarrow{CC}$ means that the edge is traversed in the opposite direction, from a junior category to a senior category).

As usual, a^* denotes a sequence of the form $\underbrace{a, a, \ldots, a}_{n}$, with $n \geq 0$, so, for example, a path of type $(\overrightarrow{CC})^*$ represents a chain of categories in the $\subseteq$ relation

As a consequence of Definitions 14 and 15, the relation $c_1 \subseteq c_2$ between categories c_1 and c_2 is represented in the policy graph by a path of type $(\overrightarrow{CC})^*$.

Note that an edge may have type $\overrightarrow{CC}$ and also type $\overleftarrow{CC}$: if $le(e).\mathsf{target} = \{v_i, v_{i+1}\}$ then the edge can be traversed in both directions and has both types. In fact, a chain of equivalent categories is represented by a path $v_0, v_1, \ldots, v_n$ of both types $(\overrightarrow{CC})^*$ and $(\overleftarrow{CC})^*$, instead of using, for instance, a cycle $v_0, v_1, \ldots, v_n, v_0$ of type $(\overrightarrow{CC})^*$ — our goal is to avoid redundant edges (Definition 17) in well-formed policy graphs.

We now give an example of a path illustrating the propagation of authorisations.

Example 16 The following image shows a constrained path of type $PC, \overrightarrow{CC}, \overrightarrow{CC}, CA, AR$:

Definition 17 (Redundant edges) Redundant edges of type PC, CC and CA are defined as follows:

- An edge of type PC, between nodes representing a principal p and a category c is *redundant* if there is a path of size $n \geq 2$ and type $PC, (\overrightarrow{CC})^*$ connecting p and c in the graph.
- An edge of type CC, between nodes representing two categories c_1 and c_2 is *redundant* if there is a path of size $n \geq 2$ and type $(\overrightarrow{CC})^*$ connecting c_1 and c_2 in the graph.
- An edge of type CA between nodes representing a category c and an action a is *redundant* if there is a path of size $n \geq 2$ and type $(\overrightarrow{CC})^*CA$ connecting c and a in the graph.

The definition of redundant edge takes into account the fact that senior categories inherit permissions from junior categories, therefore there is no need to establish connections to senior categories, if there is already a connection through a junior category. In the case of edges of type CC, transitive edges are redundant.

Definition 18 (Well-formed policy graph) A policy graph $(\mathcal{V}, E, lv, le)$ is *well-formed* iff for every $v_1, v_2 \in \mathcal{V}$, if $lv(v_1).\mathsf{ent} = lv(v_2).\mathsf{ent}$ and $lv(v_1).\mathsf{type} = lv(v_2).\mathsf{type}$ (that is, $v_1 \equiv v_2$) then $v_1 = v_2$; for every $e_1, e_2 \in E$, if $le(e_1).\mathsf{adj} = le(e_2).\mathsf{adj} = \{v_1, v_2\}$, then $e_1 = e_2$; and every $e \in E$ with $le(e).\mathsf{adj} = \{v_1, v_2\}$ satisfies one of the following conditions:

(a) $lv(v_1).\mathsf{type} = P \wedge lv(v_2).\mathsf{type} = C$. This corresponds to an edge of type PC, which connects a principal and a category.
(b) $lv(v_1).\mathsf{type} = C \wedge lv(v_2).\mathsf{type} = A_{\mathcal{R}}$. This corresponds to an edge of type CA, which connects a category and an action.
(c) $lv(v_1).\mathsf{type} = A_{\{r_1,\ldots,r_n\}} \wedge lv(v_2).\mathsf{type} = R \wedge lv(v_2).\mathsf{ent} = r_i$. This corresponds to an edge of type $A_{\{r_1,\ldots,r_n\}}R$, which connects an action and a resource.
(d) $lv(v_1).\mathsf{type} = C \wedge lv(v_2).\mathsf{type} = C$ and $le(e).\mathsf{target} \subseteq \{v_1, v_2\}$, for an edge connecting categories, that is, an edge of type CC.
(e) There are no redundant edges.

Example 19 The graph shown in Example 12 is a well-formed policy graph.

Proposition 20 (Characterisation of paths) Any path of size 3 in a well-formed policy graph, starting in a node of type P and ending in a node of type R, must have the following shape:

More generally, in a well-formed policy graph, paths starting in a node of type P and ending in a node of type R must start with an edge of type PC and end with edges of type CA and AR.

Proof Direct consequence of the restriction imposed on edge types in well-formed policy graphs (see Definition 18). 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 or a node of type C, 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. Longer paths starting in a node of type P and ending in a node of type R must traverse nodes of type C before traversing an edge of type AC and an edge of type AR □

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.

Definition 21 (Relations $\mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA}_\mathcal{G}$ and $\mathcal{PAR}_\mathcal{G}$ associated with $\mathcal{G}$) Let $\mathcal{G}$ be a well-formed policy graph. Then we define the following relations associated with $\mathcal{G}$:

- $\mathcal{PCA}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}) \mid \mathsf{type}(v_1, v_2) = PC\}$.
- $\mathcal{ARCA}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}, lv(v_3).\mathsf{ent}) \mid \mathsf{type}(v_1, v_2, v_3) = CA, AR\}$.
- $\mathcal{PAR}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid \exists v_{21}, \ldots, v_{2n}$ *s.t.* $\mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = PC, (\overrightarrow{CC})^*, CA, AR\}$.

We denote $CBAC_\mathcal{G}$ the tuple $\langle \mathcal{E}_\mathcal{G}, \mathcal{PCA}_\mathcal{G}, \mathcal{ARCA}_\mathcal{G}, \mathcal{PAR}_\mathcal{G} \rangle$, where $\mathcal{E}_\mathcal{G} = (\mathcal{P}_\mathcal{G}, \mathcal{C}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{R}_\mathcal{G}, \mathcal{S}_\mathcal{G}, \subseteq)$ consists of the sets of principals, categories, actions, resources, situational identifiers, and the containment relation $\subseteq$, obtained from the graph. For example, $\mathcal{P}_\mathcal{G} = \{lv(v).\mathsf{ent} \mid lv(v).\mathsf{type} = P, v \in \mathcal{V}\}$. The other sets in $\mathcal{E}_\mathcal{G}$ are computed similarly, and the relation $\subseteq$ can be obtained directly from the graph by a simple path computation, as indicated in Definition 14. The set $\mathcal{S}$, of situational identifiers, contains all the remaining attributes (note that, $\mathcal{S}$ is usually used to denote environmental information relevant to the policy, such as time, places, etc.).

Example 22 In the policy graph $\mathcal{G}$ shown in Example 12, the relation $\mathcal{PCA}_\mathcal{G}$ contains the tuples:

(*J. Dorian*, *Intern*),	(*C. Tuck*, *Resident*),
(*P. Cox*, Specialist),	(*P. Flowers*, *Nurse Practitioner*)
(*L. Roberts*, *Registed Nurse*),	(*C. Espinosa*, *Registed Nurse*)

and the relation $\mathcal{PAR}_{\mathcal{G}}$ contains:

(*J. Dorian*, *Read*, *Lab result*),	(*C. Tuck*, *Read*, *Lab result*),
(*P. Cox*, *Read*, *Lab result*),	(*L. Roberts*, *Perform*, *Specimen collection*),
(*L. Roberts*, *Cancel*, *Lab order*),	(*C. Espinosa*, *Perform*, *Specimen collection*),
(*C. Espinosa*, *Cancel*, *Lab order*),	(*P. Flowers*, *Perform*, *Specimen collection*)

Now we are ready to show that any well-formed policy graph represents a policy, and for any CBAC policy there is (at least one) associated policy graph.

Proposition 23 For each well-formed policy graph $\mathcal{G} = (\mathcal{V}, E, lv, le)$, the tuple $CBAC_{\mathcal{G}}$ defines a CBAC policy.

Proof By Def. 2, it is sufficient to prove that $\mathcal{PCA}_{\mathcal{G}}$, $\mathcal{ARCA}_{\mathcal{G}}$ and $\mathcal{PAR}_{\mathcal{G}}$ satisfy axiom $(a1)$, which is a consequence of Definitions 21 and 14. □

Regarding the converse, note that there may be more than one graph that generates a given 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, which contains one node for each principal, category, action and resource, one edge for each tuple in the relations $\mathcal{PCA}$ and $\mathcal{ARCA}$, and edges for $\subseteq$.

Proposition 24 For any static CBAC policy $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR} \rangle$ there exists a well-formed policy graph $\mathcal{G}$ such that $\mathcal{PCA} = \mathcal{PCA}_{\mathcal{G}}$, $\mathcal{ARCA} = \mathcal{ARCA}_{\mathcal{G}}$,$\mathcal{PAR} = \mathcal{PAR}_{\mathcal{G}}$.

Proof The set of nodes in $\mathcal{G}$ is determined by the set $\mathcal{E}$ of entities in the CBAC policy: For each principal $p \in \mathcal{P}$, there is a node v such that $lv(v).\mathsf{ent} = p$ and $lv(v).\mathsf{type} = P$, since we assume principals are uniquely identified by their name and type. Similarly, for each resource $r \in \mathcal{R}$ there is a node v such that $lv(v).\mathsf{ent} = r$ and $lv(v).\mathsf{type} = R$. For actions we need to consider the resources to which they apply and identify actions that have the same name and apply to the same resources. More precisely, for each action $a \in \mathcal{A}$, let $\{(\mathcal{C}_1, \mathcal{R}_1), \ldots, (\mathcal{C}_k, \mathcal{R}_k)\}$, be pairs of sets of categories and resources, respectively, such that, for each pair $(\{c_1, \ldots, c_n\}, \ldots, \{r_1, \ldots, r_m\})$, the tuple (a, r_i, c_j) is in $\mathcal{ARCA}$, for every $1 \leq i \leq m$, $1 \leq j \leq n$, and for every r, c such that $(a, r, c) \in \mathcal{ARCA}$ then $c \in \mathcal{C}_i, r \in \mathcal{R}_i$ for some $1 \leq i \leq k$. Furthermore, for any two pairs $(\mathcal{C}_i, r \in \mathcal{R}_i)$, $(\mathcal{C}_j, r \in \mathcal{R}_j)$, with $i \neq j$, then $\mathcal{C}_i \neq \mathcal{C}_j \wedge \mathcal{R}_i \neq \mathcal{R}_j$. Then for each $1 \leq i \leq k$, with $\mathcal{C}_i = \{c_{i1}, \ldots, c_{in_i}\}$, $\mathcal{R}_i = \{r_{i1}, \ldots, r_{im_i}\}$ there exists a node v representing the action a, such that $lv(v).\mathsf{ent} = a \wedge lv(v).\mathsf{type} = A_{\mathcal{R}_i}$ and edges $e_{11}, \ldots, e_{1n}, e_{21}, \ldots, e_{2m}$, such that $lv(e_{1j}).\mathsf{adj} = \{v_{c_{ij}}, v\}$, $1 \leq j \leq n$, $lv(e_{2l}).\mathsf{adj} = \{v, v_{r_{il}}\}, 1 \leq l \leq m$.

3.2. Representing static CBAC policies with prohibitions

To represent prohibitions we need to be able to define the relations $\mathcal{BARCA}_{\mathcal{G}}$ and $\mathcal{BAR}_{\mathcal{G}}$ in addition to $\mathcal{ARCA}_{\mathcal{G}}$ and $\mathcal{PAR}_{\mathcal{G}}$. This means that, whenever we have an edge connecting a node of type C with a node of type A, we must specify if this corresponds to an authorisation or a prohibition. It also means that to deal with CBAC policies with prohibitions we need *multigraphs*, where more than one edge may connect two given nodes. Note that a policy could have both kinds of edges between categories and actions and still satisfy the axiom $(a4)$, i.e., the overall policy may still be consistent (e.g. if no principal belongs to the category). However, it is likely that the allocation of both an authorisation and a prohibition for the same action to a category is not intended, so it will be useful to be able to detect this kind of situation.

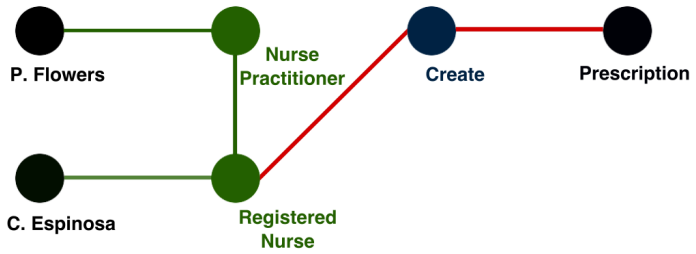
To deal with policies including prohibitions, we consider an extra field auth on labels of edges of type CA, with values in $\{A, B\}$, to represent positive and negative authorisations (or authorised and banned actions), respectively. We also annotate edge types CA with the same possible values $\{A, B\}$, and divide edges of type CA into two sets CA^A and CA^B, corresponding to edges representing authorisations and prohibitions, respectively. Edges are no longer uniquely identified by their adjacent nodes, but rather by adj and auth (unless auth is undefined for a particular edge)[3].

The definition of well-formed policy graph has to be adapted to accommodate prohibitions. Our aim is to be able to compute the policy relations directly from the graph. The relation $\mathcal{PCA}_{\mathcal{G}}$ can be computed as in Definition 21, but the computation of $\mathcal{BAR}_{\mathcal{G}}$ involves traversing edges between categories in the inverse sense (since a category inherits prohibitions from its senior categories, whereas it inherits authorisations from junior categories). For this reason, we define a notion of *constrained inverse path* (differing from Definition 13 only in the traversal of edges between categories).

Definition 25 A *constrained inverse path* of length n in a policy graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, e_1, v_1, e_2 \ldots, e_n, v_n$, such that for all $1 \leq i \leq n$, $le(e_i).\mathsf{adj} = \{v_{i-1}, v_i\} \wedge v_{i-1} \in le(e_i).\mathsf{target}$ if target exists in $le(e_i)$.

Example 26 The following figure shows the propagation of the prohibition to "create a prescription", represented by the red edge: it is propagated from the Registered Nurse to the Nurse Practitioner (if the former is not allowed to create prescriptions then the latter is not allowed either).

[3]We use the notation $e_1 \equiv e_2$ if the edges have the same values for adj and auth (or just for adj if auth is undefined).



Constrained paths between nodes of type C and constrained inverse paths between nodes of type C are characterised by types of the form $(\overrightarrow{CC})^*$ and $(\overleftarrow{CC})^*$ respectively. This characterisation is useful to compute the relations $\mathcal{PAR}_\mathcal{G}$ and $\mathcal{BAR}_\mathcal{G}$ associated with a graph $\mathcal{G}$. First, we define the conditions under which a policy graph with prohibitions is well formed.

Note that, as before, we can define redundant edges of type CA (see Definition 17), but now we need to consider separately the edges of type CA^A and CA^B: an edge of type CA^A connecting a category c with an action a is redundant if there is already a path of type $(\overrightarrow{CC})^*CA^A$ connecting c and a (this is because senior categories inherit permissions from junior categories); an edge of type CA^B is redundant if there is another path of type $(\overleftarrow{CC})^*CA^B$ connecting the same nodes. This is because junior categories inherit prohibitions from senior categories, so there is no need to add the edge of type CA^B from a junior category to an action if a senior category is already connected to this action.

Definition 27 (Well-formed policy graph with prohibitions) A policy graph $(\mathcal{V}, E, lv, le)$ is a *well-formed policy graph with prohibitions* iff

1. For every $v_1, v_2 \in \mathcal{V}$, if $lv(v_1).\mathsf{ent} = lv(v_2).\mathsf{ent}$ and $lv(v_1).\mathsf{type} = lv(v_2).\mathsf{type}$ (that is, $v_1 \equiv v_2$) then $v_1 = v_2$.
2. Every $e \in E$, where $le(e).\mathsf{adj} = \{v_1, v_2\}$, satisfies one of the following conditions:
   (a) $lv(v_1).\mathsf{type} = P \wedge lv(v_2).\mathsf{type} = C$. This corresponds to an edge of type PC, which connects principals to categories.
   (b) $lv(v_1).\mathsf{type} = C \wedge lv(v_2).\mathsf{type} = A$. This corresponds to an edge e of type CA, which connects categories to actions, and in this case $le(e).\mathsf{auth}$ must be defined, such that $le(e).\mathsf{auth} \in \{A, B\}$.
   (c) $lv(v_1).\mathsf{type} = A_{\{r_1,\ldots,r_n\}} \wedge lv(v_2).\mathsf{type} = R \wedge lv(v_2).\mathsf{ent} = r_i$. This corresponds to an edge of type $A_{\{r_1,\ldots,r_n\}}R$, which connects an action and a resource.
   (d) $lv(v_1).\mathsf{type} = C \wedge lv(v_2).\mathsf{type} = C$ and $le(e).\mathsf{target} \subseteq \{v_1, v_2\}$. This corresponds to an edge of type CC, which connects categories.

   Moreover, for every $e_1, e_2 \in E$, if $le(e_1).\mathsf{adj} = le(e_2).\mathsf{adj} = \{v_1, v_2\}$, then either $e_1 = e_2$ or e_1, e_2 have type CA and $le(e_1).\mathsf{auth} \neq le(e_2).\mathsf{auth}$.

3. If a constrained path and an inverse constrained path start in the same node p of type P and end in nodes of type A, such that the last edges in the paths are of type CA^A, CA^B respectively, then the end nodes must be different.
4. There are no redundant edges.

From a well-formed policy graph with prohibitions, we can extract a CBAC policy with prohibitions.

Definition 28 The relations $\mathcal{PCA}_\mathcal{G}$, $ARCA_\mathcal{G}$, $\mathcal{BARCA}_\mathcal{G}$, $\mathcal{PAR}_\mathcal{G}$, $\mathcal{BAR}_\mathcal{G}$ and $\mathcal{UNDET}_\mathcal{G}$ associated with a policy graph with prohibitions, $\mathcal{G}$, are defined as follows.

- $\mathcal{PCA}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}) \mid \mathsf{type}(v_1, v_2) = PC\}$.
- $\mathcal{ARCA}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}, lv(v_3).\mathsf{ent}) \mid \mathsf{type}(v_3, v_1, v_2) = CA^A, AR\}$.
- $\mathcal{BARCA}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}, lv(v_3).\mathsf{ent}) \mid \mathsf{type}(v_3, v_1, v_2) = CA^B, AR\}$.
- $\mathcal{PAR}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid \exists v_{21}, \ldots, v_{2n}\ s.t.\ \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = PC, (\overrightarrow{CC})^*, CA^A, AR\}$.
- $\mathcal{BAR}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid \exists v_{21}, \ldots, v_{2n}\ s.t.\ \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = PC, (\overleftarrow{CC})^*, CA^B, AR\}$.
- $\mathcal{UNDET}_\mathcal{G} = \{(lv(v_1).\mathsf{ent}, lv(v_2).\mathsf{ent}, lv(v_3).\mathsf{ent}) \mid lv(v_1).\mathsf{type} = P, lv(v_2).\mathsf{type} = A, lv(v_3).\mathsf{type} = R\} - (\mathcal{PAR}_\mathcal{G} \cup \mathcal{BAR}_\mathcal{G})$.

As before we write $CBAC_\mathcal{G}$ as an abbreviation for the tuple
$$\langle \mathcal{E}, \mathcal{PCA}_\mathcal{G}, \mathcal{ARCA}_\mathcal{G}, \mathcal{PAR}_\mathcal{G}, \mathcal{BARCA}_\mathcal{G}, \mathcal{BAR}_\mathcal{G}, \mathcal{UNDET}_\mathcal{G} \rangle$$
where the sets of principals, categories, actions, resources and situational identifiers, as well as the relation $\subseteq$, are defined as indicated in Definition 21.

Considering the relations above, the results in Propositions 23 and 24 can be extended to graphs/policies with prohibitions.

Proposition 29 For each well-formed policy graph with prohibitions $\mathcal{G}$, the tuple $CBAC_\mathcal{G}$ defines a CBAC policy with prohibitions.

Proof To prove the result, we need to show that the relations $\mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA}_\mathcal{G}$, $\mathcal{PAR}_\mathcal{G}$, $\mathcal{BARCA}_\mathcal{G}$ and $\mathcal{BAR}_\mathcal{G}$ and $\mathcal{UNDET}_\mathcal{G}$ given in Definition 28 satisfy axioms $(a1)$-$(a4)$, according to Def. 4. It is easy to see that axioms $(a1)$-$(a3)$ are satisfied. To show $(a4)$, that is, $\mathcal{PAR}_\mathcal{G} \cap \mathcal{BAR}_\mathcal{G} = \emptyset$, we rely on condition 3 in Definition 27. □

Proposition 30 For any static CBAC policy with prohibitions $\langle \mathcal{E}, \mathcal{PCA}, \mathcal{ARCA}, \mathcal{PAR}, \mathcal{BARCA}, \mathcal{BAR}, \mathcal{UNDET} \rangle$ there exists a well-formed policy graph with prohibitions $\mathcal{G}$ such that $\mathcal{PCA} = \mathcal{PCA}_\mathcal{G}$, $\mathcal{ARCA} = \mathcal{ARCA}_\mathcal{G}$, $\mathcal{PAR} = \mathcal{PAR}_\mathcal{G}$, $\mathcal{BARCA} = \mathcal{BARCA}_\mathcal{G}$, $\mathcal{BAR} = \mathcal{BAR}_\mathcal{G}$, $\mathcal{UNDET} = \mathcal{UNDET}_\mathcal{G}$.

3.3. Representing Distributed CBAC Policies

We now extend the notion of policy graph to accommodate multi-site policies, by including site information in the labels.

Definition 31 (Distributed policy graph) Let $\mathcal{G} = (\mathcal{V}, E, lv, le)$ be a well-formed policy graph with prohibitions, representing a CBAC 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 $(\mathcal{V}, E, lv', le')$, where $lv'(v) = \mathsf{update}(lv(v), \mathsf{site}, \{s\})$, for all $v \in \mathcal{V}$ and $le'(e) = \mathsf{update}(le(e), \mathsf{site}, \{s\})$, for all $e \in E$.

A distributed policy graph is a tuple $(\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n}, \mathcal{OP})$ where $\mathcal{G}_{s_i} = (\mathcal{V}_i, E_i, lv'_i, le'_i)$ is the policy graph of site s_i, $1 \leq i \leq n$, and $\mathcal{OP}$ is an operation on graphs that will be used to define the global policy as a composition of site policies. For each location $s \in \mathcal{S}$, the relations $\mathcal{PCA}_s$, $\mathcal{ARCA}_s$, $\mathcal{BARCA}_s$, $\mathcal{PAR}_s$ and $\mathcal{BAR}_s$ are defined (as paths on $\mathcal{G}_s$) as in the non-distributed scenario (cf. Def. 28). We call $\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n}$ the *site graphs*.

Note that the entities of the metamodel (principals, categories, actions and resources) may 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. This information is given by the site field in the labels.

Formalising the global policy in terms of $\mathcal{PAR}$ and $\mathcal{BAR}$, is not so straightforward. One possibility to define $\mathcal{PAR}$ is simply by the union of the site relations, but there are more sophisticated ways to combine the policies. This is the purpose of the $\mathcal{OP}$ parameter in the distributed policy graph.

We now define some useful operations on policy graphs. We will define these as binary operations, but they can be easily generalised to the n-ary case.

We assume without loss of generality that in any two site graphs the sets of nodes (resp. edges) are disjoint (for example, if an entity, let's say a principal p, is known in more than one site, each site policy graph will have its own node v to represent this principal). This does not contradict Def. 27: in each site policy graph, nodes are uniquely identified by the values of the ent and type fields in their records.

Definition 32 (Union Graph) Given two well-formed site graphs $\mathcal{G}_1 = (\mathcal{V}_1, E_1, lv_1, le_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, E_2, lv_2, le_2)$, where $lv_i.(v_i).\mathsf{site} = \{s_i\}$, $i = 1, 2$, the union graph $\mathcal{G}_1 \cup \mathcal{G}_2$ is a graph $(\mathcal{V}, E, lv, le)$ where:

$\mathcal{V}$ is the smallest set satisfying:

1. for every $v_1 \in \mathcal{V}_1$ such that there exists no $v_2 \in \mathcal{V}_2$ where $v_1 \equiv v_2$, $v_1 \in \mathcal{V}$ and $lv(v_1) = lv_1(v_1)$;
2. similarly, for every $v_2 \in \mathcal{V}_2$ such that there exists no $v_1 \in \mathcal{V}_1$ where $v_1 \equiv v_2$, $v_2 \in \mathcal{V}$ and $lv(v_2) = lv_2(v_2)$;
3. for every pair $v_1 \in \mathcal{V}_1$, $v_2 \in \mathcal{V}_2$ such that $v_1 \equiv v_2$, there exists a node $v \in \mathcal{V}$, which we call the *image of* v_1 *and* v_2 (written $v = Im(v_i)$, $i = 1, 2$) such that $lv(v).\mathsf{ent} = lv_1(v_1).\mathsf{ent}$, $lv(v).\mathsf{type} = lv_1(v_1).\mathsf{type}$, $lv(v).\mathsf{site} = \{s_1, s_2\}$ and for every other pair of attribute and value $(a_{1i} = v_{1i}) \in$

$lv_1(v_1)$ (respectively, $(a_{2j} = v_{2j}) \in lv_2(v_2)$), $lv(v).a_{1i}^{s_1} = v_{1i}$ (respectively, $lv(v).a_{2j}^{s_2} = v_{2j}$), that is, the record $lv(v)$ is obtained by merging the records $lv_1(v_1)$, $lv_2(v_2)$, keeping track of the sites where the attributes were originally defined by means of superindices;

and E is the smallest set satisfying:

1. for every $e_1 \in E_1$ such that $le_1(e_1).\mathsf{adj} = \{v_1, v_2\}$ and there is no $e_2 \in E_2$ where $le_2(e_2).adj = \{v_3, v_4\}$ and $\{v_1, v_2\} \equiv \{v_3, v_4\}$, $e_1 \in E$ and $le(e_1) = le_1(e_1)$;
2. similarly, for every $e_2 \in E_2$ such that $le_2(e_2).\mathsf{adj} = \{v_1, v_2\}$ and there is no $e_1 \in E_1$ where $le_1(e_1).adj = \{v_3, v_4\}$ and $\{v_1, v_2\} \equiv \{v_3, v_4\}$, $e_2 \in E$ and $le(e_2) = le_2(e_2)$;
3. for every pair $e_1 \in E_1$, $e_2 \in E_2$ such that $le_1(e_1).\mathsf{adj} = \{v_1, v_2\}$, $le_2(e_2).\mathsf{adj} = \{v_3, v_4\}$, $\{v_1, v_2\} \equiv \{v_3, v_4\}$ and $le_1(e_1).\mathsf{auth} = le_2(e_2).\mathsf{auth}$ (i.e., the attribute auth is defined and equal in both, or undefined in both), there exists an edge $e \in E$, where $le(e).\mathsf{adj} = \{Im(v_1), Im(v_2)\}$, $le(e).\mathsf{auth} = le_1(e_1)$ if defined, $le(e).\mathsf{site} = \{s_1, s_2\}$, and for every other $(a_{1i} = v_{1i}) \in le_1(e_1)$ (respectively, $(a_{2j} = v_{2j}) \in le_2(e_2)$), $le(e).a_{1i}^{s_1} = v_{1i}$ (respectively, $le(e).a_{2j}^{s_2} = v_{2j}$);
4. for every pair $e_1 \in E_1$, $e_2 \in E_2$ such that $le_1(e_1).\mathsf{adj} = \{v_1, v_2\}$, $le_2(e_2).\mathsf{adj} = \{v_3, v_4\}$, $\{v_1, v_2\} \equiv \{v_3, v_4\}$ and $le_1(e_1).\mathsf{auth} \neq le_2(e_2).\mathsf{auth}$ (i.e., the attribute auth is defined in both, but one edge corresponds to an authorisation and the other to a prohibition, no other cases are possible since the site graphs are well formed), there exist two edges $e_1' \in E$, $e_2' \in E$, such that $le(e_1').\mathsf{adj} = le(e_2').\mathsf{adj} = \{Im(v_1), Im(v_2)\}$, $le(e_1').\mathsf{site} = \{s_1\}$, $le(e_2').\mathsf{site} = \{s_2\}$, and for every other $(a_{1i} = v_{1i}) \in le_1(e_1)$ (respectively, $(a_{2j} = v_{2j}) \in le_2(e_2)$), $le(e_1').a_{1i}^{s_1} = v_{1i}$ (respectively, $le(e_2').a_{2j}^{s_2} = v_{2j}$).

We can define the intersection graph $\mathcal{G}_1 \cap \mathcal{G}_2$ in a similar way, merging nodes and edges if they appear in both graphs and discarding them otherwise.

The difference graph $\mathcal{G}_1 \setminus \mathcal{G}_2$ is the graph $(\mathcal{V}_1, E, lv_1, le)$, obtained by eliminating the edges from $\mathcal{G}_1$ that appear in $\mathcal{G}_2$, that is, for every $e_1 \in E_1$, such that $le_1(e_1).\mathsf{adj} = \{v_1, v_2\}$, if there is $e_2 \in E_2$, such that $e_1 \equiv e_2$ then $e_1 \notin E$, otherwise $e_1 \in E$ and $le(e_1) = le_1(e_1)$.

The union, intersection and difference graph have different applications: whereas the union graph is useful for combining policies, the intersection can be used to detect redundancies and the difference to remove redundancies.

From a union graph we can extract relations $\mathcal{PAR}_{\mathcal{G}}$ and $\mathcal{BAR}_{\mathcal{G}}$ defining a composition of site policies according to different kinds of union operators (e.g., union with priority to grant or union with priority to deny; see Example 6). To extract the policy relations, we start by classifying paths according to the site graphs from which they originate.

Definition 33 A *site-path* of length n in site s, in a union graph $\mathcal{G}$, between two nodes v_0, v_n, is a sequence $v_0, e_1, v_1, e_2 \ldots, e_n, v_n$, such that $le(e_i).\mathsf{adj} = \{v_{i-1}, v_i\}$ for all $1 \leq i \leq n$, and $s \in le(e_i).\mathsf{site}$, for all $1 \leq i \leq n$, that is, all the edges in the path belong to the same site s.

To highlight the fact that all the edges in the path belong to the same site, the type of a site-path will be annotated with the site identifier, for example, $(PC, CA^A, AR)^s$ denotes a site-path of length 3 in site s.

Now we can compute the authorisations and prohibitions of a distributed CBAC policy defined as a union with priority to grant (or as a union with priority to deny, or as a *"first-applicable"*) by computing site-paths: for example, site-paths in the union graph with type $PC, (\overrightarrow{CC})^*, CA^A, AR$ represent authorisations (since such a path in the union graph indicates the authorisation exists in at least one policy).

Definition 34 Let $\mathcal{G} = \mathcal{G}_1 \cup \ldots \cup \mathcal{G}_n$ be a union graph obtained from site policies $\mathcal{G}_1, \ldots, \mathcal{G}_n$. The relations $\mathcal{PAR}^G_{\mathcal{G}}$ and $\mathcal{BAR}^G_{\mathcal{G}}$ (resp. $\mathcal{PAR}^D_{\mathcal{G}}$ and $\mathcal{BAR}^D_{\mathcal{G}}$) defining union with priority to grant (resp. union with priority to deny) and $\mathcal{PAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}}$, $\mathcal{BAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}}$ for "first applicable" are defined as follows.

- $\mathcal{PAR}^G_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overrightarrow{CC})^*, CA^A, AR)^s\}.$
- $\mathcal{BAR}^G_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overleftarrow{CC})^*, CA^B, AR)^s \wedge$
  $(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \notin \mathcal{PAR}^G_{\mathcal{G}}\}.$
- $\mathcal{BAR}^D_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overleftarrow{CC})^*, CA^B, AR)^s\}.$
- $\mathcal{PAR}^D_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overrightarrow{CC})^*, CA^A, AR)^s \wedge$
  $(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \notin \mathcal{BAR}^D_{\mathcal{G}}\}.$
- $\mathcal{PAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s_i, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overrightarrow{CC})^*, CA^A, AR)^{s_i} \wedge$
  $\neg\exists j, v'_{21}, \ldots, v'_{2m}\ s.t.\ (j < i) \wedge$
  $\mathsf{type}(v_1, v'_{21}, \ldots, v'_{2m}, v_3, v_4) = (PC, (\overleftarrow{CC})^*, CA^B, AR)^{s_j}\}.$
- $\mathcal{BAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}} = \{(lv(v_1).\mathsf{ent}, lv(v_3).\mathsf{ent}, lv(v_4).\mathsf{ent}) \mid$
  $\exists v_{21}, \ldots, v_{2n}, s_i, \mathsf{type}(v_1, v_{21}, \ldots, v_{2n}, v_3, v_4) = (PC, (\overleftarrow{CC})^*, CA^B, AR)^{s_i} \wedge$
  $\neg\exists j, v'_{21}, \ldots, v'_{2m}\ s.t.\ (j < i) \wedge$
  $\mathsf{type}(v_1, v'_{21}, \ldots, v'_{2m}, v_3, v_4) = (PC, (\overrightarrow{CC})^*, CA^A, AR)^{s_j}\}.$

Proposition 35 Let $\mathcal{G} = \mathcal{G}_1 \cup \ldots \cup \mathcal{G}_n$ be a union graph and $CBAC_{\mathcal{G}_i} = \langle \mathcal{E}_i, \mathcal{PCA}_i, \mathcal{ARCA}_i, \mathcal{BARCA}_i, \mathcal{PAR}_i, \mathcal{BAR}_i, \mathcal{UNDET}_i \rangle$, for each $1 \leq i \leq n$, be the policies associated to each site graph according to Definition 21. Let $Pol = \langle \mathcal{E}, \{\mathcal{PCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{ARCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{BARCA}_i\}_{i\in\mathcal{S}}, \{\mathcal{PAR}_i\}_{i\in\mathcal{S}}, \{\mathcal{BAR}_i\}_{i\in\mathcal{S}}, \{\mathcal{UNDET}_i\}_{i\in\mathcal{S}}, \mathcal{OP}_{par}, \mathcal{OP}_{bar} \rangle$, where $\mathcal{E}$ is obtained as the union of the corresponding sets (of principals, categories, etc) in $\mathcal{E}_i$, and $\mathcal{OP}_{par}, \mathcal{OP}_{bar}$ define a union with priority to grant (resp. union with priority to deny, "first-applicable" composition). Then Pol is a distributed CBAC policy, and relations $\mathcal{PAR}^G_{\mathcal{G}}$ and $\mathcal{BAR}^G_{\mathcal{G}}$ (resp. $\mathcal{PAR}^D_{\mathcal{G}}$ and $\mathcal{BAR}^D_{\mathcal{G}}$, $\mathcal{PAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}}$, $\mathcal{BAR}^{FA(s_1,\ldots,s_n)}_{\mathcal{G}}$), respectively compute its set of permissions and prohibitions.

Proof According to Definition 7, we need to prove that Pol satisfies axioms $(b1)$-$(h1)$, where $\mathcal{OP}_{par}, \mathcal{OP}_{bar}$ define a union operator with priority to grant (resp. priority to deny or first applicable).

Axioms $(b1)$-$(e1)$ hold because each site policy is well-formed. The interesting axioms are $(f1)$, $(g1)$ and $(h1)$.

To show $(f1)$ and $(g1)$ for union with priority to grant (resp. union with priority to deny, first applicable) we need to prove that the authorisations computed from the union graph correspond to these operators. Let us consider first the case of union with priority to grant. We need to show that if one of the site policies authorises p to perform action a on r, then the global policy authorises it; and if at least one policy prohibits p to perform a on r and no policy authorises it then the global policy denies it. This follows from the fact that $\mathcal{PAR}^G_{\mathcal{G}}$ contains the tuple (p, a, r) if there is a site path corresponding to an authorisation (p, a, r) for some site s, and $\mathcal{BAR}^G_{\mathcal{G}}$ contains the tuple (p, a, r) only if it is not in $\mathcal{PAR}^G_{\mathcal{G}}$ (so, no site authorises it) and moreover at least one site prohibits it.

To show $(h1)$, we observe that by Definition 34, $\mathcal{PAR}^G_{\mathcal{G}}$ and $\mathcal{BAR}^G_{\mathcal{G}}$ are disjoint since $\mathcal{BAR}^G_{\mathcal{G}}(p, a, r)$ implies $\neg\mathcal{PAR}^G_{\mathcal{G}}(p, a, r)$.

We reason in a similar way for the union with priority to deny and first-applicable. □

3.4. Representing Dynamic CBAC Policies

Relations between entities in the CBAC 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 snapshot of the authorisations and prohibitions that hold in the system at a particular time. From the graph one can extract the relations $\mathcal{PCA}$, $\mathcal{ARCA}$, $\mathcal{PAR}$ and $\mathcal{BAR}$ at a particular instant, but not how to build the next "photo". For this, we will enrich policy graphs by adding mechanisms to specify dynamic policies. More precisely, we will use functional attributes to model the evolution of authorisations and prohibitions in the system.

Definition 36 (Dynamic policy graph) A *dynamic policy graph* is a well-formed policy graph with prohibitions $\mathcal{G} = (\mathcal{V}, E, lv, le)$ such that for every $v \in \mathcal{V}$

there exists a field fun in $lv(v)$ such that $lv(v).\mathsf{fun} = R$, for some convergent rewrite system R satisfying the following conditions:

- if $lv(v).\mathsf{type} = P$, then $lv(v).\mathsf{fun}$ defines a function pca that returns the list of categories of the principal $lv(v).\mathsf{ent}$, written $\mathsf{pca}_{lv(v).\mathsf{fun}} = [c_1, \ldots, c_n]$, which may depend on the system state;
- if $lv(v).\mathsf{type} = C$, then $lv(v).\mathsf{fun}$ defines two functions arca and barca, each returning a list of pairs of the form *(action,resource)* (one of permissions of the category $lv(v).\mathsf{ent}$ and the other of prohibitions, respectively), written $\mathsf{arca}_{lv(v).\mathsf{fun}} = [(a_{11}, r_{11}), \ldots, (a_{1n}, r_{1n})]$ (resp. $\mathsf{barca}_{lv(v).\mathsf{fun}} = [(a_{21}, r_{21}), \ldots, (a_{2m}, r_{2m})]$, which may depend on the system state.

In this definition, we assume that the functions that compute categories, permissions and prohibitions have access to a global state, which could, for instance, be a log containing the history of events that are relevant for the system.

All dynamic policy graphs are well formed by definition, and therefore define a unique CBAC policy by Proposition 23. However, since now policy graphs include function specifications, we need to check that the current graph structure is consistent with these specifications.

Definition 37 A dynamic policy graph $\mathcal{G}$ is *correct* with respect to a system state st (i.e., at a particular instant) iff:

- for every node v of type P, $\mathsf{pca}_{lv(v).\mathsf{fun}} = [c_1, \ldots, c_n]$ iff there exists an edge $e_i \in E$, with $le(e_i).\mathsf{adj} = \{v, v_i\}$ for $i = 1, \ldots, n$, such that $lv(v_i).\mathsf{ent} = c_i$;
- for every node v of type C,

  $\mathsf{arca}_{lv(v).\mathsf{fun}} = [(a_{1i}, r_{1i}) | i = 1, \ldots, n]$, $\mathsf{barca}_{lv(v).\mathsf{fun}} = [(a_{2j}, r_{2j}) | j = 1, \ldots, m]$

  iff there exists in E edges e_{1i}, e'_{1i}, with $le(e_{1i}).\mathsf{adj} = \{va_{1i}, vr_{1i}\}$, $le(e'_{1i}).\mathsf{auth} = A$ and $le(e'_{1i}).\mathsf{adj} = \{v, va_{1i}\}$ for $i = 1, \ldots, n$ such that $lv(va_{1i}).\mathsf{ent} = a_{1i}$, $lv(vr_{1i}).\mathsf{ent} = r_{1i}$, and edges e_{2j}, e'_{2j}, with $le(e_{2j}).\mathsf{adj} = \{va_{2j}, vr_{2j}\}$, $le(e'_{2j}).\mathsf{auth} = B$ and $le(e'_{2j}).\mathsf{adj} = \{v, va_{2j}\}$ for $j = 1, \ldots, m$ such that $lv(va_{2j}).\mathsf{ent} = a_{2j}$, $lv(vr_{2j}).\mathsf{ent} = r_{2j}$.

If a policy graph $\mathcal{G}$ is correct with respect to a state st, we say that the *configuration* $\langle \mathcal{G}, st \rangle$ is correct.

A correct dynamic policy graph represents a category-based policy where the relations evolve as the system state changes. Formally, we define a transition relation on configurations.

Definition 38 Let $\rightarrow$ denote a transition relation on system states, that is, $st \rightarrow st'$ denotes a transition from state st to st'. The transition relation on configurations is defined as follows: $\langle \mathcal{G}_i, st_i \rangle \rightarrow_{ev} \langle \mathcal{G}_{i+1}, st_{i+1} \rangle$ if $\langle \mathcal{G}_i, st_i \rangle$ is a correct configuration, $\mathcal{G}_{i+1}$ is a policy graph with the same set of nodes as $\mathcal{G}_i$, $st_i \rightarrow st_{i+1}$ and $\mathcal{G}_{i+1}$ is correct with respect to st_{i+1}.

Note that, each correct configuration defines its own set of $\mathcal{PAR}$ and $\mathcal{BAR}$ relations, as explained in Definition 28. Thus, dynamic policy graphs can be used to define the evolution of authorisations and prohibitions. This dynamic behavior can potentially be defined as graph rewriting rules, using a formalism capable of representing graphs with relevant information associated to nodes/edges. In the future, our goal is to use the formalism of port-graphs and the PORGY graphical framework [4] to implement dynamic policy graphs.

4. Analysis of Category-Based Policies

Managing access control policies is a challenging task, in particular in any medium size organisation, where several policies are usually in place and their interactions need to be controlled. This motivated the work on distributed and federative access control models, and the development of tools and technologies to manage the policies (see, e.g., [14] for a survey of analysis techniques). Security administrators need to query policies to extract information and to verify basic correctness properties. In this section we first consider standard queries in CBAC policies and show how they can be answered using policy graphs. We then show how to use policy graphs to manage and analyse policies.

4.1. Extracting information from policy graphs and checking constraints

According to the basic policy analysis model defined in [14], analysis queries are classified as *policy metadata queries*, *policy content queries* and *policy effect queries*.

Policy metadata queries concern metadata information about the policy; examples of information that can be retrieved with metadata queries are the author and date of creation of the policy, and the available principals, actions and resources in the system. Using the graph model we propose in this paper, metadata information about principals, actions and resources can be extracted by a simple traversal of the graph, since this kind of information is stored in the labels of nodes and edges. If metadata concerning the whole policy, such as date of creation, is of interest, a simple solution would be to add a node of type 'metadata' in the graph to store additional information.

Policy content queries directly examine the content of policies, and policy effect queries are queries about the outcome the policy will produce in various situations. More precisely, policy effect queries relate to the authorisations and prohibitions specified by the policy, and usually mention principals, actions and resources. Property verification queries are a particular case of policy effect queries.

Policy content queries and policy effect queries are the traditional administrator queries in access control systems. We discuss them below.

Policy content queries. The following are typical policy content queries in CBAC; they aim at obtaining information about the policy at the time the

query is issued.[4]

- Q1: Are all the principals associated with at least one category?
- Q2: Are there (positive or negative) permissions associated to each category?
- Q3: Are all the resources accessible (in terms of principals and permissions)?
- Q4: For a given category, who are the associated principals?
- Q5: To which categories belongs a given principal?
- Q6: For a given category, what are the associated permissions?
- Q7: For a given principal, what are the associated permissions?

Following [14], if for a given policy there is a principal not assigned to any category, or a category without associated permissions, or a resource which is not accessible, we say that the policy is *ineffective* for that principal, category, or resource respectively. It is unlikely that a policy is intentionally written to behave in this way; the analysis should try to identify ineffectiveness. We show how this can be done (and can be easily visualised) using the graph-based framework defined in the previous sections. We deal with non-distributed systems first and then generalise the results to distributed policies.

Given a policy represented by a well-formed graph $\mathcal{G} = (\mathcal{V}, E, lv, le)$, the queries Q1-Q7 above can be formalised and answered in the following way, where the semantics of the $\subseteq$ relation between categories is set inclusion (if a principal p is in the category c_1 and $c_1 \subseteq c_2$ then p is in c_2; see Definition 1).

1. All the principals are associated with at least one category if and only if the degree of every node of type P is positive.
2. All the categories have some associated (positive or negative) permissions if and only if for each node v of type C there is a path of type $(\overrightarrow{CC})^*, CA^A, AR$ or a path of type $(\overleftarrow{CC})^*, CA^B, AR$ starting in v .
3. All the resources are accessible if and only if for each node v of type R there is a path of type $PC, (\overrightarrow{CC})^*, CA^A, AR$ starting in v.
4. To retrieve the set of principals that belong to a category c, compute the set $\{v_1, \ldots, v_n\}$ of nodes of type P such that for each v_i there is a path of type $PC, (\overrightarrow{CC})^*$ starting from v_i and ending in the node of type C representing c, and output $lv(v_i).\mathsf{ent}$ $(1 \leq i \leq n)$.

[4] If a policy is static (e.g., in the case of an RBAC policy) then the answer does not depend on time. However, in the general case of a dynamic policy (e.g., policies in the DEBAC model) the answer may depend on events that happen in the system, and so it may change in time.

5. For a given node v of type P representing the principal p, the set of 'least' categories to which p belongs is obtained by computing the set of neighbours of type C of v. To retrieve all the categories to which p belongs, it is sufficient to compute the set of paths of type $PC, (\overrightarrow{CC})^*$ starting from v.
6. For a given node v of type C representing the category c, the set of associated (positive or negative) permissions is obtained by computing all the paths of type $(\overrightarrow{CC})^*, CA^A, AR$ and all the paths of type $(\overleftarrow{CC})^*, CA^B, AR$, starting at v. The last two nodes of each path define a permission (authorisation or prohibition) associated to that category.
7. For a given node v of type P representing the principal p, the set of associated (positive or negative) permissions is obtained by computing all the paths of type $PC, (\overrightarrow{CC})^*, CA^A, AR$ and all the paths of type $PC, (\overleftarrow{CC})^*, CA^B, AR$, starting at v. The last two nodes of each path define a permission associated to that principal.

Proposition 39 The queries Q1-Q7 listed above can be answered in polynomial time with respect to $|V| + |E|$.

Proof In each case, the algorithm suggested consists of a simple graph traversal. □

The queries mentioned above are still valid in a distributed scenario, and can be solved in a similar way, either by considering paths in the individual graphs $\mathcal{G}_{s_1}, \ldots, \mathcal{G}_{s_n}$, or computing paths in the union-graph defined above (Definition 32).

Policy effect queries. Policy effect queries relate to specific properties of the policy and are usually stated in terms of authorisations, prohibitions and their interactions. Typical queries relate to the *totality* and *consistency* of the policy [19]. A policy is total if every access request from a principal p to perform an action a on a resource r receives a grant or deny answer. It is consistent if there is a unique answer for each request (this ensures that the same request cannot be both authorised and prohibited by the policy). The latter is a challenging problem, in particular in distributed systems, where the answer to an access request depends on the way the individual policies are composed. Policy graphs can help in the analysis of distributed policies, since they provide a visual specification of the policy.

We consider first totality and consistency of policies defined by a single policy graph, and then deal with distributed policies: To check that a CBAC policy is total at a given time, assuming there is a well-formed policy graph $\mathcal{G}$ representing the policy, it is sufficient to compute the relations $\mathcal{PAR}_\mathcal{G}$ and $\mathcal{BAR}_\mathcal{G}$ as specified in Definition 28 and check that $\mathcal{PAR}_\mathcal{G} \cup \mathcal{BAR}_\mathcal{G} = \mathcal{P} \times \mathcal{A} \times \mathcal{R}$. Consistency is guaranteed by axiom $(a4)$, that is, the policy graph satisfies $\mathcal{PAR}_\mathcal{G} \cap \mathcal{BAR}_\mathcal{G} = \emptyset$.

In a distributed policy, where the global access rights are defined in terms of individual policies, the consistency of the global policy relies on a specific mechanism to avoid conflicts between permissions in different sites (e.g., union with

priority to deny). In the previous section, we proposed to define a distributed policy graph as a tuple of site policies, and then compute a graph operation, depending on the kind of composition defined by the distributed policy. To analyse a distributed CBAC policy we first compute a distributed policy graph by computing the policy graphs of each individual CBAC policy, and then compute a graph using the operators to combine authorisations and permissions specified in the distributed policy. Let $\mathcal{G}$ be the resulting graph. Totality and consistency of the distributed policy can now be checked on $\mathcal{G}$, by computing the $\mathcal{PAR}_{\mathcal{G}}$ and $\mathcal{BAR}_{\mathcal{G}}$ relations, as specified in Definition 34, and then proceeding as in the case of a single policy.

The checks described above can answer a totality or a consistency query for a static policy, or a query pertaining to a dynamic policy at a specific time (i.e. for a given configuration consisting of a system state and a policy graph). For dynamic policies, these properties are generally undecidable, since it is necessary to consider the way the relations $\mathcal{PCA}$, $\mathcal{ARCA}$ and $\mathcal{BARCA}$ evolve. Decidable sufficient conditions exist, if these relations are defined by rewrite rules; we refer to [19] (see also [21]) for details. These results can be directly applied to dynamic policy graphs (see Definition 37).

Policy effect queries include also queries about specific properties of policies, which involve checking that the policy satisfies certain constraints. A well-known example is the "separation of duty" constraint, where no user should be allowed to perform two conflicting actions on the same resource (e.g., issue a purchase order and approve it).

Proposition 40 Let $\mathcal{G}$ be a well-formed policy graph and assume an action a_1 in a resource r is in conflict with an action a_2 in r. The policy graph $\mathcal{G}$ ensures separation of duty constraint, if for each principal p, in the paths of type $PC, (\overrightarrow{CC})^*, CA^A, AR$ (resp. $(PC, (\overrightarrow{CC})^*, CA^A, AR)^s$ for distributed policies) linking the nodes representing p and r, the set of values for the attribute ent in labels of nodes of type A in each path do not contain simultaneously a_1 and a_2.

Another interesting constraint in RBAC systems is the *mutual exclusion* of roles. Such a constraint between roles r_1 and r_2 is satisfied if no user can be a member of both roles. RBAC policies are a particular case of static CBAC policies, and mutual exclusion constraints can be specified as constraints on paths in the policy graph as follows.

Proposition 41 Let $\mathcal{G}$ be a policy graph representing an RBAC policy, where roles r_1 and r_2 are defined as categories under a mutual exclusion constraint. To check that the policy satisfies this mutual exclusion constraint, it is sufficient to apply the algorithm specified to answer Q4 above (For a given category, who are the associated principals?). More precisely, for the given roles, compute the associated sets of principals and check that the sets are disjoint.

5. Application: Emergency Policies as Distributed CBAC Policies

In this section we consider a particular kind of 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.

Emergency situations are usually associated with one or more events that happen in the system [24]. In this approach, events can be thought of as elementary or compound actions [31], which we represent using terms of the form $\mathsf{event}(e_i, p, a, o, t, l)$, following [17]. 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 with 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 [24]:

> *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 medical 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:

$$\forall P,\ (P, patient) \in \mathcal{PCA} \ \Rightarrow\ (read, record(P), doctor(P)) \in \mathcal{ARCA}_{\pi_1}$$
$$\forall P,\ (P, patient) \in \mathcal{PCA} \ \wedge\ emerg(bcrd, P) \Rightarrow (read, record(P), doctor) \in \mathcal{ARCA}_{\pi_2}$$

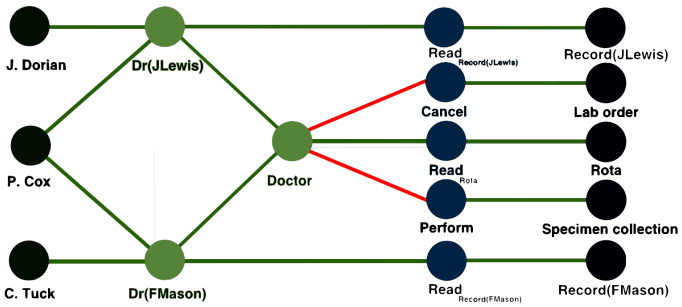
Operationally, we specify rewrite rules for arca in the standard (π_1) and emergency (π_2) sites, and combine the policies using a union operator with priority

to grant, or a first-applicable(π_2, π_1) if there are both positive and negative authorisations to combine.

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

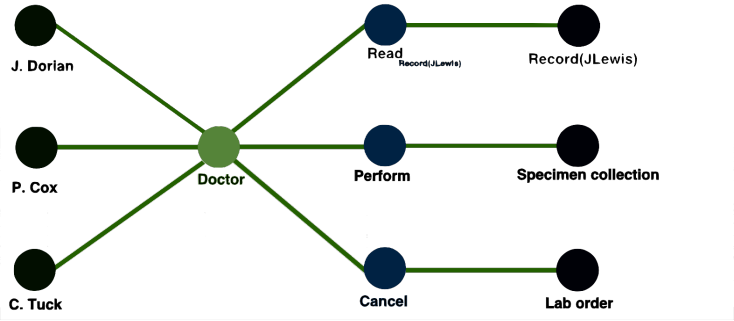
where $patientList$ returns the list of patients, that is, P such that $patient \in \mathsf{pca}(P)$.

In this scenario, the graph representing the standard policy, $\mathcal{G}_{\pi_1}$, has a node v of type P (principal) for each doctor, where $lv(v).ent$ contains the doctor's identification, and nodes of type C representing the categories $doctor(p)$, for each patient p (if the number of nodes is too large, zooming mechanisms could be implemented to help security administrators visualise the subgraphs of interest, see Section 6). A category node $doctor(p)$ is connected to nodes representing the authorised actions for p's doctors (e.g., an action $read$ with type $A_{record(p)}$, on p's patient record represented by a node $record(p)$ of type R). Furthemore, every category node $doctor(p)$ is connected to a category node $doctor$ by an edge of type $\overrightarrow{CC}$ (that is, $doctor(p) \subseteq doctor$), to which all the doctors are connected. The category node $doctor$ is connected to the actions that are available to all doctors (e.g., reading the rota information). In the following image representing the standard policy graph, red edges are used to represent banned actions.



The dynamic policy graph representing the emergency policy in site π_2, $\mathcal{G}_{\pi_2}$, has again a node of type P to represent each doctor, linked to a node of type C representing the category $doctor$, with the function arca_{π_2} defined above. This function takes into account the occurrence of the event $emerg(bcrd, p)$: there is an edge connecting the category $doctor$ to $read$ and $record(p)$ if and only if p

is suffering a cardiac emergency.



We now discuss 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.

The graph representation of a policy in an emergency scenario will be given by combining the graphs of the normal policy and the emergency policy within a union graph (Def. 32) and implementing a "union with priority to grant" operator or a "first-applicable" operator (see Def. 34). The analysis described in the previous 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 $PC, (\overrightarrow{CC})^*, CA, AR$ of the form $p, c^*, read, record(i)$ 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, determining what are the permissions revoked by the emergency can be realised by checking for multi-edges of type CA in the union graph (corresponding to positive and negative authorisations, which shows that the normal policy and the emergency policy disagree); what permissions were created by the emergency can be obtained from the difference between the emergency and the normal policy graphs; etc. All these properties can easily be established using our graph formalisation. "Separation of duties" constraints are also useful in this context. 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)"*, can be expressed as a separation of duties and checked as explained in Section 4.

6. Graphical tools to analyse policies

In this section we describe two tools to analyse policies based on our graph formalisation.

6.1. Policy Manager

An application called *Policy Manager* [44] 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 [29]: 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 such as unassigned principals, unused resources, policy conflicts, etc. The application provides security administrators with a complete view of a policy as a tree, allowing them 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 policy queries.

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. 36), 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. 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).

6.2. The `G-ACM` *Tool*

Another prototype based on our formalism, called the `G-ACM` Tool [48], uses a rule engine to implement the dynamic aspects of categories (more precisely, the the JBoss Drools rule engine [28]). In `G-ACM` , rules are used both to compute permissions from given relations ($\mathcal{PCA}$ and $\mathcal{ARCA}$), but also permissions that depend on customisable facts of the system.

This tool is a step towards our long-term goal of having a user-friendly language to describe the dynamic behaviour of categories, suitable for policy administrators and that could be translated into other programming languages for integration with policy analysis tools. In `G-ACM` , Drools are used to compute permissions based on a set of pre-defined rules that can be changed by the security administrator. In particular, the definitions of categories, therefore the permissions, can be changed by redefining the rules based on the available attributes.

There is a separation in the graphical representation given in Section 3, between the graphical representation of the policy at a given moment, which

corresponds to a static graph where one can use static analysis to extract properties, and the rewrite-based operational semantics that allows for the permissions to change dynamically. In `G-ACM` this separation is also clear, where Drools is used as a mechanism to derive new instances of the graph, when changes occur in the attributes defining categories.

Additional functionalities that improve permission analysis include: identifying/displaying the changes in the policy graph, resulting from two different sets of facts/parameters; filtering entities so that only the associated policy (sub-) graph is displayed; aggregate nodes into compound nodes, which can be unfolded whenever required, representing sets of entities with the same connections (this results in more compact and intelligible graphs, allowing the identification of entities with the same properties).

7. Related Work

Previous works on CBAC policies have used only textual languages and have focused mainly on the expressivity of the model, the analysis of policies and the techniques that can be used to enforce policies [9, 17, 18, 19, 1]. Graph-based languages have been used in the literature to model and analyse access control problems, but not within CBAC (with the exception of [3], which is the basis for this paper and considers CBAC policies without prohibitions). Koch et al. [38, 39], inspired by the role-permission assignment model proposed by Nyanchama and Osborn [46], use directed graphs to formalise RBAC, using a graph model similar to Baldwin's privilege graphs [8]. A privilege graph is a three layered acyclic graph where the first layer represents the users, the second the roles and the third the permissions (actions and resources). A distinctive feature of [38, 39] is the modelling of role management operations by graph transformation rules. The graphs used in [38, 39] are typed in a similar way as our policy graphs, and are also labelled, but labels in [38, 39] are simply identifiers to encode RBAC policies, whereas we use a richer label structure in order to express richer CBAC policies where access rights may depend on data associated to entities in the policy. Using our policy graphs we can represent the RBAC policies considered in [38, 39], since a role is a particular case of a category; however, the graphs used in [38, 39] represent also session information. We have not dealt with sessions in policy graphs in this paper, but the notion of a session in CBAC is similar to the notion of a session in RBAC and the representation of sessions provided in [38, 39] could be easily adapted to policy graphs representing CBAC policies.

LASCO [33] is a language in which system states and security constraints can be specified using graphs, which are labelled with attributes and values. A LASCO policy graph describes both the situation (system state) in which an access control policy applies and the access control constraints. More precisely, a system state is represented by a graph where nodes represent users and resources and edges represent actions (events in the LASCO terminology). The edges are oriented from user (source) to object (target). Nodes and edges are labelled with attributes and values, and the constraints are annotated on edges in the form

of predicates involving attributes and values. In our policy graphs, actions are represented by nodes, which can also be labelled with attributes and values, and the LASCO constraints can be seen as a particular case of category specification: the predicates and matching semantics of LASCO can be easily mapped to rewrite rules defining the pca function in the CBAC model (see Definition 3). Another closely related system is Miró [32], which is similar to LASCO in that graphs are used to represent instances of the system and constraints, however, Miró focuses on security policies for file systems, whereas we consider general CBAC policies.

Another related approach uses term rewrite rules to model particular access control models, relying on the power of term rewriting systems to express general dynamic access control policies [10, 49, 37, 20, 19]. Properties of policies are then checked using techniques to check confluence and termination of sets of rewrite rules. Our approach combines the use of a visual graph formalism to represent a concrete state of the system, and the use of rewrite rules to model the dynamics of the system. In [37], it is shown how narrowing can be used to solve administrator queries of the form "what if a request is made under these conditions?", by representing a query as a pair of a term and a first-order equational constraint. Narrowing-based techniques could also be used in dynamic graph policies, since the functions defining the main relevant relations are defined by sets of rewrite rules.

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 [49, 23, 15] 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 time, events, and histories that are not included as elements of *RT* or RBAC. In [21], 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.

Several extensions of RBAC have been proposed to deal with dynamic policies, where permissions may change depending on internal or external conditions such as time, location, or context-based properties (see, for example, [25, 36, 40]). All of these extensions can be seen as instances of the more abstract CBAC model.

The Generalised TRBAC model [36] and ASL [34] are related to CBAC in that they 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 [41] provides a general framework specialised for defining specific policy requirements (in terms of credentials). Li and Tripunitara [42] state a family of security analysis problems in RBAC systems, and propose techniques to maintain desirable security properties, in particular when administrators perform operations that change the authorisations in the sys-

tem. Since the CBAC model permits the specification of dynamic categories, authorisations in a CBAC policy can change without the intervention of the administrator. The techniques developed in [42] to control administrative changes in RBAC (e.g., to ensure that roles do not grow or do not shrink) could be adapted to control the allocation of principals to categories in CBAC. This is left for future work.

Recent work on enforcement of CBAC policies within web-based applications relies on static analysis techniques and code injection [1]. Graph-based policies could be used to bridge the gap between security administrators (who are not always programming experts) and programmers applying this kind of language-based techniques for policy enforcement.

The framework that we have described is more expressive than any of the Datalog-based languages that have been proposed for distributed access control policies (see [7, 35, 26, 12]); these languages, being based on a monotonic semantics, are not especially well suited for representing dynamically changing distributed policies. Recent research addressed these limitations by extending Datalog to include notions such as updates and persistency, but these features have an operational semantics outside Datalog, which may cause semantic ambiguities [43, 45]. Dedalus [2] is an extension of Datalog with an explicit notion of time: all the predicates have an additional parameter to represent time. In this way, Dedalus can easily express the dynamics of distributed systems. However, although dealing with time explicitly is useful in certain scenarios, it is unnecessary in other cases (a simpler ordering of events might be sufficient).

Analysis of dynamic policies is also addressed by Dougherty, Fisler and Krishnamurthi [27] using Datalog to specify policies as logic programs, and state transition systems to model program execution and the passage of time.

In [13], the constraint logic programming language SecPal is proposed, to specify 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.

8. 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: from single to distributed, and from static to dynamics policies.

Using this representation, we rely on graph properties and graph algorithms, to extract properties on policies, including policy content and policy effect queries. Although most of these properties are static, 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), and are therefore significantly more challenging. Other more challenging properties that we wish to address in the future, deal with optimisation queries, such

as what is the minimal number of changes or the minimal connections necessary to ensure a particular authorisation assignment. 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 [4], a tool that allows users to visualise and simulate systems via port-graph rewriting.

Additionally, in the context of an analysis application such as the Policy Manager and the `G-ACM` tool, we would like to continue our efforts 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, 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, Hossein Mirzapour-Aghdaghi for implementing the Policy Manager tool, and João Sá for implementing the `G-ACM` tool.

[1] A. Ali and M. Fernández. Hybrid Enforcement of Category-Based Access Control. In *Proc. STM 2014 (in conjunction with ESORICS 2014)*. LNCS, Springer, 2014.

[2] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier and R. Sears. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010. Revised Selected Papers*, LNCS, 6702, pages 262–281, Springer, 2010.

[3] S. Alves and M. Fernández. A framework for the analysis of access control policies with emergency management. In *Proc. LSFA 2014, Workshop on Logical Frameworks with Applications*, ENTCS 312: 89-105, 2015.

[4] O. Andrei, M. Fernández, H. Kirchner, G. Melançon, O. Namet and B. Pinaud. Porgy: Strategy-driven interactive transformation of graphs. In *TERMGRAPH*, volume 48 of *EPTCS*, pages 54–68, 2011.

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

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

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

[8] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pp.116,132, 1990.

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

[10] 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.

[11] S. Barker. Action-status access control. In V. Lotz and B. M. Thuraisingham, editors, *Proc. ACM Int. Conf. SACMAT 2007*, pages 195–204. ACM Press, 2007.

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

[13] 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.

[14] E. Bertino, C. Brodie, S. Calo, L. F. Cranor, C. Karat, J. Karat, L. Ninghui, D. Lin, J. Lobo, Q. Ni, P.R. Rao and X. Wang. Analysis of privacy and security policies. *IBM Journal of Research and Development*, vol.53, no.2, pp.3:1,3:18, 2009.

[15] C. Bertolissi and M. Fernández. A rewriting framework for the composition of access control policies. In *Proc. PPDP 2008 - Int. Symposium on Principles and Practice of Declarative Programming*. ACM Press, 2008.

[16] 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.

[17] C. Bertolissi and Maribel Fernández. Category-based authorisation models: operational semantics and expressive power. In *Proc. ESSOS 2010*, LNCS. Springer, 2010.

[18] C. Bertolissi and M. Fernández. Rewrite specifications of access control policies in distributed environments. In *Proc. STM 2010, Int. Workshop on Security and Trust Management*, number 6710 in LNCS. Springer, 2011.

[19] C. Bertolissi and M. Fernández. A Metamodel of Access Control for Distributed Environments: Applications and Properties. *Information and Computation*, volume 238, pp. 187 – 207. Special Issue on Security and Rewriting Techniques, 2014.

[20] 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.

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

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

[23] T. Bourdier, H. Cirstea, M. Jaume and H. Kirchner. Formal specification and validation of security policies. In *FPS*, pages 148–163, 2011.

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

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

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

[27] D. Dougherty, K. Fisler and S. Krishnamurthi. Specifying and Reasoning about Dynamic Access-Control Policies. In *Proc. of IJCAR 2006*. LNCS, 4130, Springer,2006.

[28] The Drools Rule Engine. `http://www.drools.org`. Accessed: 2015-05-27.

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

[30] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[31] M. Gelfond and J. Lobo. Authorization and obligation policies in dynamic systems. In *Proc. ICLP*, pages 22–36, 2008.

[32] A. Heydon, M.W. Maimone, J.D. Tygar, J.M. Wing and A.M. Zaremski. Miró: Visual Specification of Security. *IEEE Transactions on Software Engineering*, vol. 16, no. 10, pp. 1185-1197, 1990.

[33] J. A. Hoagland. Specifying and Implementing Security Policies Using LaSCO, the Language for Security Constraints on Objects. Ph.D. dissertation, UC Davis, Computer Science, 2000.

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

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

[36] 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.

[37] C. Kirchner, H. Kirchner and A. Santana de Oliveira. Analysis of Rewrite-Based Access Control Policies. *Electronic Notes in Theoretical Computer Science* 234:55–75, 2009.

[38] M. Koch, L. V. Mancini and F. Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.* 5, 3, 332-365, 2002.

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

[40] D. Kulkarni and A. Tripathi. Context-aware role-based access control in pervasive computing systems. In *Proc. SACMAT 2008*, pages 113–122, ACM press, 2008.

[41] 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.

[42] N. Li and M. Tripunitara. *Security Analysis in Role-Based Access Control. ACM Trans. Inf. Syst. Secur.* 9(4):391–420, 2006.

[43] Y. Mao. On the declarativity of declarative networking. *Operating Systems Review.* 43(4): 19–24, 2009.

[44] 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`

[45] J. A. Navarro Pérez and A. Rybalchenko. Operational Semantics for Declarative Networking. In Proceedings of *PADL 2009*, LNCS, 5418, pages 76–90, Springer, 2009.

[46] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.* 2, 1 (February 1999), 3-33, 1999.

[47] OASIS, eXtensible Access Control Markup Language (XACML). urlhttp://www.oasis-open.org/xacml/docs/, 2003.

[48] J. Sá, S. Alves and S. Broda The `G-ACM` Tool: using the Drools Rule Engine for Access Control Management *Submitted for publication*, 2015.

[49] A. Santana de Oliveira. *Réécriture et Modularité pour les Politiques de Sécurité*. PhD thesis, Université Henri Poincaré, Nancy, France, 2008.

[50] K. Sohr, M. Drouineaud, G.-J. Ahn and M. Gogolla. Analyzing and managing role-based access control policies. *IEEE Trans. Knowl. Data Eng.*, 20(7):924–939, 2008.