A Modular Approach to Composing Access Control Policies*

Piero Bonatti
Dip. Scienze dell'Informazione
Polo di Crema
Università di Milano
Via Bramante 65
26013 Crema - Italy
bonatti@iago.crema.unimi.it

Sabrina De Capitani di Vimercati
Dip. Elettronica
Università di Brescia
Via Branze 38
25123 Brescia - Italy
decapita@ing.unibs.it

Pierangela Samarati
Dip. Scienze dell'Informazione
Polo di Crema
Università di Milano
Via Bramante 65
26013 Crema - Italy
samarati@dsi.unimi.it

ABSTRACT

Despite considerable advancements in the area of access control and authorization languages, current approaches to enforcing access control are all based on monolithic and complete specifications. This results limiting when restrictions to be enforced come from different input requirements, possibly under the control of different authorities, and where the specifics of some requirements may not even be known a priori. Turning individual specifications into a coherent policy to be fed into the access control system requires a nontrivial combination and translation process.

We address the problem of combining authorization specifications that may be independently stated, possibly in different languages and according to different policies. We propose an algebra of security policies together with its formal semantics and illustrate how to formulate complex policies in the algebra and reason about them. We also illustrate a translation of policy expressions into equivalent logic programs, which provide the basis for the implementation of the language.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design

Keywords

Access control, policy composition, algebra, logic programs

*The work reported in this paper was partially supported by the European Community within the Fifth (EC) Framework Programme under contract IST-1999-11791 – FASTER project.

CCS '00, Athens, Greece.

1. INTRODUCTION

Recent years have witnessed considerable work on access control models and languages. Many approaches have been proposed to increase expressiveness and flexibility of authorization languages by supporting multiple policies within a single framework [3, 5, 7, 8, 10, 18]. All these proposals, while based on powerful languages able to express different policies, assume a single monolithic specification of the entire policy. Such an assumption does not fit many real world situations, where access control might need to combine restrictions independently stated that should be enforced as one. As an example, consider a data warehouse collecting data from different sources, where each data source may impose access restrictions on its data; access restrictions can be stated in different languages and with reference to different paradigms. As another example, consider a large organization composed of different departments and divisions, each of which can independently specify security policies; the global policy of the organization results from the combination of all these components. A further example is provided by recent laws concerning privacy issues. In a modern information system, the security policy of the organization should combine internally specified constraints with externally imposed privacy regulations [2]. Finally, as security policies become more sophisticated, even within a single system ruled by one administrator it may be desirable to formulate the policy incrementally by assembling small, manageable, and independently conceived modules.

Existing frameworks represent these situations by translating and merging the different component policies into a single "program" in the adopted language. While existing languages are flexible enough to obtain the desired combined behavior, this method has several drawbacks. First, the translation process is far from being trivial; it must be done very carefully to avoid undesirable side effects due to interference between the component policies. Interference may result in the combined specifications not reflecting correctly the intended restrictions. Second, after translation it is not possible anymore to operate on the individual components and maintain them autonomously. Third, existing approaches cannot take into account incomplete policies, where some components are not (completely) known a priori (e.g., when somebody else is to provide that component).

This situation calls for a policy composition framework by which different component policies can be integrated while retaining their independence. In this paper, we propose an algebra for combining security policies together with its formal semantics. Complex policies are formulated as expressions of the algebra. Our framework results flexible and keeps compound specifications simple by organizing them into different levels of abstraction. The formal framework can be used to reason about properties of (possibly incomplete) specifications. We finally illustrate a translation of algebra expressions into equivalent logic programs, which provide the basis for the implementation of the language.

To our knowledge ours is the first proposal addressing composition of authorization specifications. Previous work on composition (e.g., [1, 6]) focussed on the secure behavior of program modules. Closest work lies in proposals targeted to the development of a uniform framework to express possibly heterogeneous policies [3, 7, 8, 10, 18]. However, as already discussed none of these proposals addressed composition. Our work is complementary to these proposals, which can be used to specify the individual component policies in our framework.

2. CHARACTERISTICS OF A COMPOSITION FRAMEWORK

A first step in the definition of a framework for composing policies is the identification of the characteristics that it should have. In particular, we have identified the following:

1. *Heterogeneous policy support* The composition framework should be able to combine policies expressed in arbitrary languages and enforced by different mechanisms. For instance, a data warehouse may collect data from different data sources where the security restrictions autonomously stated by the sources and associated with the data may be stated with different specification languages, or refer to different paradigms (e.g., open vs closed policy).
2. *Support of unknown policies* It should be possible to account for policies which may be partially unknown, or be specified and enforced in external systems. These policies are like "black-boxes" for which no (complete) specification is provided, but that can be queried at access control time. Think, for example, of a situation where given accesses are subject to a policy P enforcing "central administration approval". While P can respond yes or no to each specific request, neither the description of P, nor the complete set of accesses that it allows might be available. Run-time evaluation is therefore the only possible option for P. In the context of a more complex and complete policy including P as a component, the specification could be partially compiled, leaving only P (and its possible consequences) to be evaluated at run time.
3. *Controlled interference* Policies cannot always be combined by simply merging their specifications (even if they are formulated in the same language), as this could have undesired side effects causing the accesses granted or denied to not correctly reflect the specifications. As a simple example, consider the combination of two systems P_{closed}, which applies a closed policy, based on rules of the form "*grant access if* $(s, o, +a)$", and P_{open} which applies an open policy, based on rules of the form "*grant access if* $\neg(s, o, -a)$". Merging the two specifications would cause the latter decision rule to derive all authorizations not blocked by P_{open}, regardless of the contents of P_{closed}. Similar problems may arise from uncontrolled interaction of the derivation rules of the two specifications. Besides, if the adopted language is a logic language with negation, the merged program might not be stratified (which may lead to ambiguous or undefined semantics).
4. *Expressiveness* The language should be able to conveniently express a wide range of combinations (spanning from minimum privileges to maximum privileges, encompassing priority levels, overriding, confinement, refinement, etc.) in a uniform language. The different kinds of combinations must be expressed without changing the input specifications and without ad-hoc extensions to authorizations (like those introduced to support priorities). For instance, consider a department policy P_1 regulating access to documents and the central administration policy P_2. Assume that access to administrative documents can be granted only if authorized by both P_1 *and* P_2. This requisite can be expressed in existing approaches only by explicitly extending all the rules possibly referred to administrative documents to include the additional conditions specified by P_2. Among the drawbacks of this approach is the rule explosion that it would cause and the complex structure and loss of control over the two specifications which, in particular, cannot be maintained and managed autonomously anymore.
5. *Support of different abstraction levels* The composition language should highlight the different components and their interplay at different levels of abstraction. This feature is important to: *i)* facilitate specification analysis and design; *ii)* facilitate cooperative administration and agreement on global policies; *iii)* support incremental specification by refinement.
6. *Formal semantics* The composition language should be declarative, implementation independent, and based on a solid formal framework. An underlying formal framework is needed to: *i)* ensure non-ambiguous behavior and *ii)* reason about policy specifications and prove properties on them [9].

3. AN ALGEBRA OF POLICIES

To make our approach generally applicable we do not make any assumption on the subjects, objects, or actions w.r.t. which accesses have to be controlled and, therefore, authorization specifications stated. We assume sets S, O, and A denoting the subjects, objects, and actions, respectively, are given. Depending on the application context and the policy to be enforced, subjects could be users or groups thereof, as well as roles or applications; objects could be files, relations, XML documents, classes, and so on.

3.1 Preliminary concepts

We start by defining authorization terms as follows.

Definition 3.1 (Authorization term). *Authorization terms are triples of the form (s,o,a), where s is a constant in* S *or a variable over* S*, o is a constant in* O *or a variable over* O*, and a is a constant in* A *or a variable over* A.

At a semantic level, a policy is defined as a set of triples as follows.

DEFINITION 3.2 (POLICY). *A policy is a set of ground authorization terms.*

The triples in a policy P state the accesses permitted by P. Intuitively, a policy represents the outcome of an authorization specification, where, for composition purposes, it is irrelevant how specifications have been stated and their outcome computed.

The algebra (among other operations) allows policies to be restricted (by posing constraints on their authorizations) and closed w.r.t. inference rules. The model should be compatible with a variety of languages for constraining authorizations and formulating rules (e.g., [7, 8, 10, 18]). For this purpose, we make our algebra parametric w.r.t. the following languages and their semantics:

1. An *authorization constraint language* $\mathcal{L}_{acon}$ and a semantic relation $\mathsf{satisfy} \subseteq (\mathsf{S} \times \mathsf{O} \times \mathsf{A}) \times \mathcal{L}_{acon}$; the latter specifies for each ground authorization term (s, o, a) and constraint $c \in \mathcal{L}_{acon}$ whether (s, o, a) satisfies c.

2. A *rule language* $\mathcal{L}_{rule}$ and a semantic function $\mathsf{closure} : \wp(\mathcal{L}_{rule}) \times \wp(\mathsf{S} \times \mathsf{O} \times \mathsf{A}) \to \wp(\mathsf{S} \times \mathsf{O} \times \mathsf{A})$;[1] the latter specifies for each set of rules R and ground authorizations P which authorizations are derived from P by R.

For simplicity, we consider a single authorization constraint language $\mathcal{L}_{acon}$ and a single rule language $\mathcal{L}_{rule}$. Our model can be straightforwardly extended to handle many such languages simultaneously, so that compound policies can be assembled using different tools.

To fix ideas and make concrete examples, in this paper we shall adopt the following simple languages for constraints and rules:

1. $\mathcal{L}_{acon}$ contains simple binary constraints of the form $(s\ \mathsf{op}\ s_0)$, $(o\ \mathsf{op}\ o_0)$, or $(a\ \mathsf{op}\ a_0)$, where *i)* s, o, a are variables ranging over S, O, and A, respectively; *ii)* op can be $\leq, \geq, <, >, =$, where disequalities model hierarchical relationships among subjects, objects, and actions (e.g., file/directory; user/group; role/superrole) [8], and *iii)* s_0, o_0, a_0 are members of S, O, and A, respectively. Moreover, $\mathcal{L}_{acon}$ contains unary constraints of the form $p(s)$, $p(o)$, $p(a)$, where s, o, a are as before and p belongs to a fixed but arbitrary set of predicates that evaluate properties associated with subjects, objects, and actions. Such predicates, together with the above binary ones, will be called *base predicates*.

2. As a rule language we adopt simple Horn clauses, built from authorization terms and base predicates, of the form $(s, o, a) \leftarrow L_1 \wedge \ldots \wedge L_n$, where each L_i is either an authorization term or a logical atom of the form $(x\ \mathsf{op}\ y)$ or $p(x)$, with op and p base predicates and x and y terms of the appropriate type. Accordingly, $\mathsf{closure}(R, P)$ is defined as the least Herbrand model of the logic program $\Pi = R \cup P \cup B$, where B is the definition of the base predicates. We recall that the least Herbrand model of Π can be expressed as $T_\Pi \uparrow \omega$, where T_Π is the *immediate consequence operator* associated with Π, and $T_\Pi \uparrow \omega$ is the limit of the monotonic infinite sequence $\emptyset, T_\Pi(\emptyset), T_\Pi(T_\Pi(\emptyset)), \ldots, T^i_\Pi(\emptyset), \ldots$, with i a natural number (see [11] for more details).

[1]For all sets X, $\wp(X)$ denotes the powerset of X.

These languages have been chosen with the goal of keeping presentation as simple as possible, focusing attention on policy composition, rather than authorization properties and inference rules.

3.2 Policy expressions

We are now ready to define our algebra. First, its syntax is introduced, then the meaning of each operator will be illustrated. We assume an infinite set of *policy identifiers* is given. Policy expression syntax is given by the following BNF grammar:

$$E ::= \mathbf{id} \mid E + E \mid E \& E \mid E - E \mid E \hat{} C \mid o(E, E, E) \mid E * R \mid T(E) \mid (E)$$
$$T ::= \tau\, \mathbf{id}.T \mid \tau\, \mathbf{id}.E$$

Here **id** is the token type of policy identifiers, E is the nonterminal describing *policy expressions*, T is a construct called *template*, that represents incomplete policy expressions, C and R are the constructs describing $\mathcal{L}_{acon}$ and $\mathcal{L}_{rule}$, respectively (they are not specified here because the algebra is parametric w.r.t. $\mathcal{L}_{acon}$ and $\mathcal{L}_{rule}$).

The above syntax is disambiguated by assigning suitable precedence and associativity to each operator:

op	precedence	associativity
τ	0	non-associative
.	1	non-associative
$+, \&, -$	2	left-associative
$*, \hat{}$	3	left-associative

We now discuss the semantics of expressions of the algebra. Formally, semantics is a function that maps each expression onto a set of ground authorizations, that is, a policy. The simplest possible expressions, namely identifiers, are bound to sets of triples by *environments*.

DEFINITION 3.3 (ENVIRONMENT). *An environment e is a partial mapping from policy identifiers to sets of ground authorizations. By $e[X/S]$ we denote a modification of environment e such that*

$$e[X/S](Y) = \begin{cases} S & \text{if } Y = X \\ e(Y) & \text{otherwise} \end{cases}$$

The binding can either be stated explicitly or generated by some engine. The policy can be seen in such a case as a "black box". In symbols, the semantics of an identifier X w.r.t. an environment e will be denoted by $[\![X]\!]_e \stackrel{\text{def}}{=} e(X)$.

Note that X might be undefined in the environment; in that case $[\![X]\!]_e$ is undefined. Similarly, the semantics of compound expressions that use undefined identifiers is undefined.

Compound policies can be obtained by combining policy identifiers through the algebra operators. Let the metavariables P and P_i range over policy expressions:

Addition (+) merges two policies by returning their union. Formally, $[\![P_1 + P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \cup [\![P_2]\!]_e$. For instance, in an organization composed of different divisions, access to the main gate can be authorized by any of the administrator of the divisions (each of them knows which users need access to reach their division). The totality of the accesses through the main gate to be authorized should then be the union of the statements of each division. Intuitively, additions can be

applied in any situation where accesses can be authorized if allowed by any of the component policies.

Conjunction (&) merges two policies by returning their intersection. Formally, $[\![P_1 \& P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \cap [\![P_2]\!]_e$. While addition allows an access if any of the component policies allows it, conjunction requires all the component policies to agree on the fact that the access should be granted. Intuitively, while addition enforces maximum privilege, conjunction enforces minimum privilege. For instance, consider an organization in which divisions share certain documents (e.g., clinical folders of patients). An access to a document may be allowed only if all the authorities that have a say on the document agree on it. That is, if the corresponding authorization triple belongs to the intersection of their policies.

Subtraction (-) restricts a policy by eliminating all the accesses in a second policy. The formal definition is $[\![P_1 - P_2]\!]_e \stackrel{\text{def}}{=} [\![P_1]\!]_e \setminus [\![P_2]\!]_e$. Intuitively, subtraction specifies exceptions to statements made by a policy and encompasses the functionality of negative authorizations in existing approaches. The advantages of subtraction over explicit denials include a simplification of the conflict resolution policies and a clearer semantics. In particular, the difference operation allows to clearly and unambiguously express the two different uses of negative authorizations, namely *exceptions to positive statements* and *explicit prohibitions*, which are often confused or require explicit ad-hoc extensions to authorizations [13]. Subtraction can also be used to express different overriding/conflict resolution criteria as needed in each specific context, without affecting the form of the authorizations (c.f. Section 7).

Closure ($*$) closes a policy under a set of inference (derivation) rules. The general definition is $[\![P * R]\!]_e \stackrel{\text{def}}{=} \mathsf{closure}(R, [\![P]\!]_e)$. Derivation rules can, for example, enforce propagation of authorizations along hierarchies in the data system, or enforce more general forms of implication, related to the presence or absence of other authorizations, or depending on properties of the authorizations [7]. Intuitively, derivation rules can be thought of as logic rules whose head is the authorization to be derived and whose body is the condition under which the authorization can be derived. The closure of a policy P under a set of rules R produces a policy containing all the authorizations that can be derived by evaluating R against P according to a given semantics. Recall that, in the examples of this paper, we assume rules to be Horn clauses. The general definition is thus specialized to $[\![P * R]\!]_e = T_{R \cup [\![P]\!]_e \cup B} \uparrow \omega$.

Scoping restriction ($\hat{}$) restricts the application of a policy to a given set of subjects, objects, and actions. Formally, $[\![P \hat{} c]\!]_e \stackrel{\text{def}}{=} \{(s,o,a) \mid (s,o,a) \in [\![P]\!]_e,\ (s,o,a)\ \mathsf{satisfy}\ c\}$, where $c \in \mathcal{L}_{acon}$. Scoping is particularly useful to "limit" the statements that can be established by a policy and to enforce authority confinement. Intuitively, all authorizations in the policy which do not satisfy the scoping restriction are ignored, and therefore result ineffective. For instance, an organization can attach to a policy P_{adm}, to be specified by the administration, a scoping restriction that limits the authorizations that P_{adm} can state to administrative documents, expressed as "$o \leq$ `adm_documents`" or, equivalently, "`adm_document`(o)". Similarly, the organization can attach to policy P_{lib}, to be specified by the librarian, a scoping restriction "$a =$ `read`" that limits statements to read-only actions. Scoping restrictions can be also used to select a portion of a policy, which may be subject to a different treatment than the rest of P, for example, being overridden as discussed below.

Overriding (o) replaces part of a policy with a corresponding fragment of a second policy. The portion to be replaced is specified by means of a third policy. Formally, $[\![o(P_1,P_2,P_3)]\!]_e \stackrel{\text{def}}{=} [\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$. For instance, consider the case where users of a library who have passed the due date for returning a book cannot borrow the same book anymore *unless* the responsible librarian vouchers for (authorizes) the loan. The policy can be expressed as $o(P_{lib}, P_{vouch}, P_{block})$, where P_{lib} are the accesses authorized at the library, P_{block} is the "black-list" of accesses, and P_{vouch} are the accesses authorized by the responsible librarian. Beside being specified explicitly, the fragment of P_1 to be overridden can be specified by means of scoping restrictions selecting triples in P_1 that satisfy a given condition. For instance, consider a department where access to laboratories is regulated by policy P_{lab}. Suppose that to be admitted, non-US citizens need *also* the chair consent, stated by policy P_{chair}. In other words, the portion of P_{lab} referred to non-US citizens should be overridden by its intersection with P_{chair}. This policy is then specified as $o(P_{lab}, P_{chair}, P_{lab} \hat{} \mathsf{non{-}UScitizen}(s))$. Note the importance of substituting to the fragment the intersection of the two policies, meaning *both* of them must agree on the access. Cases in which the fragment should simply be substituted with the second policy can be achieved via difference (or scoping restriction) and addition. In the following, we will abbreviate expression $o(P_1, P_2, P_1 \hat{} c)$ as $o(P_1, P_2, \hat{} c)$.

Template (τ) defines a partially specified policy that can be completed by supplying the parameters. $[\![\tau X.P]\!]_e$ is a function over policies (ground authorization sets), such that for all policies S, $[\![\tau X.P]\!]_e(S) \stackrel{\text{def}}{=} [\![P]\!]_{e[X/S]}$. Templates can be instantiated by applying them to a policy expression. For all policy expressions P_1, $[\![(\tau X.P)(P_1)]\!]_e \stackrel{\text{def}}{=} [\![\tau X.P]\!]_e([\![P_1]\!]_e) = [\![P]\!]_{e[X/[\![P_1]\!]_e]}$. We say that all the occurrences of X in an expression $\tau X.P$ are *bound*. The *free* identifiers of a policy expression P are all the identifiers with non-bound occurrences in P. Clearly, $[\![P]\!]_e$ is defined iff all the free identifiers in P are defined in e.

Templates are useful for representing partially specified policies, where some component X is to be specified at a later stage. For instance, X might be the result of further policy refinement, or it might be specified by a different authority. When a specification P_1 for X is available, the corresponding global policy can be simply expressed as $(\tau X.P)(P_1)$. Templates with multiple parameters can be expressed and applied us-

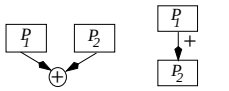
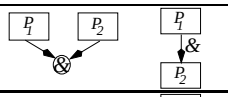
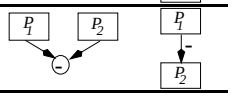
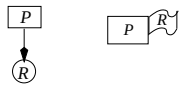
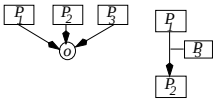
Operator	Semantics $[\![\ ]\!]_e$	Graphical representation
$P_1 + P_2$	$[\![P_1]\!]_e \cup [\![P_2]\!]_e$	
$P_1 \& P_2$	$[\![P_1]\!]_e \cap [\![P_2]\!]_e$	
$P_1 - P_2$	$[\![P_1]\!]_e \setminus [\![P_2]\!]_e$	
$P * R$	$T_{R \cup [\![P]\!]_e \cup B} \uparrow \omega$	
$P \hat{} c$	$\{t \in [\![P]\!]_e \mid t \text{ satisfy } c\}$	
$o(P_1,P_2,P_3)$	$[\![(P_1 - P_3) + (P_2 \& P_3)]\!]_e$	

Figure 1: Operators of the algebra and their graphical representation

ing the following abbreviations:
$\tau X_1, \ldots, X_n.P = \tau X_1.\tau X_2. \ldots \tau X_n.P$
$(\tau X_1, \ldots, X_n.P)(P_1, \ldots, P_n) =$
$(\ldots((\tau X_1, \ldots, X_n.P)(P_1))(P_2)\ldots)(P_n)$.

Figure 1 summarizes the operators of our algebra and their semantics and illustrates two possible graphical representations of algebraic operations. Basically, each policy is represented as a box containing the policy expression or the policy identifier. In the first representation, operators are represented as circles labeled with the operator. In the second representation, operators are represented as labels associated with arcs. Moreover, closure of a policy P under rules R is represented as a flag labeled R attached to P's box; the application of a scoping restriction c to a policy P is represented as a small box attached to P's box; and the overriding $o(P_1, P_2, P_3)$ is represented as an arc from P_1 to P_2, where the arc has attached P_3 (for simplicity, when P_3 is $P_1 \hat{} c$ only an oval labeled c is attached to the arc). The two representations can be used interchangeably as they better suit for the clarity of the resulting picture.

4. EXAMPLE SCENARIOS

We illustrate some examples of expressions stating protection requirements by composing policy statements through different operators.

Example 1: Hospital

Consider a hospital composed of three departments, namely `Radiology`, `Surgery`, and `Medicine`. Each of the departments is responsible for granting access to data under their (possibly overlapping) authority domains, where domains are specified by a scoping restriction. The statements made by the departments are then unioned, meaning the hospital considers an access as authorized if any of the department policies states so. For privacy regulations, however, the hospital will not allow any access (even if authorized by the departments) to `lab_tests` data unless there is patient consent for that, stated by policy $P_{consents}$. In terms of the algebra, the hospital policy can be represented as $o(P_{rad} \hat{} [o \leq \texttt{rad}] + P_{surg} \hat{} [o \leq \texttt{surg}] + P_{med} \hat{} [o \leq \texttt{med}], P_{consents}, \hat{} [o \leq \texttt{lab\_tests}])$. Accordingly, `lab_tests` data will be released only if *both* the hospital authorizes the release and the interested patient consents to it. As an example of component policies, let us zoom into $P_{consents}$ and P_{med}.

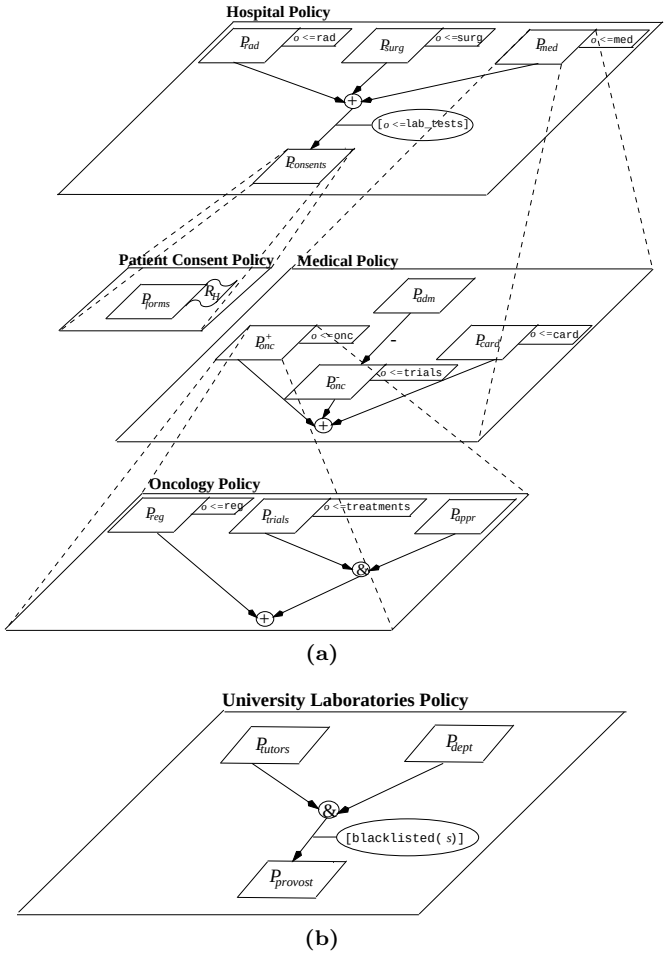


Figure 2: A policy regulating access to the hospital data (a) and to the university laboratories (b)

$P_{consents}$ reports accesses to laboratory tests for which there is patient consents. Authorizations in $P_{consents}$ are collected by the hospital administration by means of forms that patients fill in when admitted. Patients' consents can refer to single individuals (e.g., `John Doe` can individually point out that his daughter `jane.doe` can access his tests) as well as to subject classes (e.g., `research_labs` and `hospitals`), and can refer to single documents or to classes of them. Authorizations specified for subject/object classes are propagated to individual users and documents by classical hierarchy-based derivation rules (see Section 7). Denoting such rules with R_H, we can express $P_{consents}$ as $P_{forms} * R_H$.

Policy P_{med} of the medical department is composed of the policies of its two divisions, `Cardiology` and `Oncology`, and of a policy P_{adm} specified by the central administration of the department. The `Oncology` division can revoke authorizations in P_{adm} regarding data related to clinic trials, by listing them in P^{-}_{onc}. In addition, each of the divisions can specify further authorizations (policies P^{+}_{onc} and P_{card}), whose scope is restricted to objects in their respective domains. The medical department's policy P_{med} can be expressed as $P_{adm} - P^{-}_{onc}\hat{}[o \leq \texttt{trials}] + P^{+}_{onc}\hat{}[o \leq \texttt{onc}]$ $+ P_{card}\hat{}[o \leq \texttt{card}]$. Let us take a closer look at the component policy P^{+}_{onc}. P^{+}_{onc} consists of two separate policies: P_{reg}, regulating access to the hospital cancer register; and P_{trials}, regulating access to experimental cancer treatments. In addition, access to experimental cancer treatments can be allowed only if the `Cancer Clinical Trials Office` (`CCTO`) has approved testing the treatments on patients. By representing approvals as policy P_{appr}, we can write policy P^{+}_{onc} as $P_{reg}\hat{}[o \leq \texttt{reg}] + (P_{trials}\hat{}[o \leq \texttt{treatments}]\&P_{appr})$.

Figure 2(a) illustrates the global policy regulating access to the hospital data and the content of the component policies discussed.

Example 2: University Laboratories

Consider a policy regulating access to university laboratories. To use machines, students must be authorized by both the lab tutors and the department administration. The tutors and the administration can specify authorizations at different levels of details. In particular, policy P_{tutors}, specified by the tutors, can state permissions on a single user-single machine basis, with statements like `(jim.smith,machine1, login)`. The department can state permissions with reference to groups of students and machines, with statements like `(cs101,cs-lab,login)` which, closed under classical propagation rules implies a permission for all students enrolled in class `cs101` to use machines in the `cs-lab`. Authorized accesses are defined as a conjunction of the two policies (intuitively, students should have permission to use the laboratory by the department and machine assignment by the tutor). In addition, access to any laboratory resource is forbidden to students blacklisted for infraction to rules (e.g., honor code); only an explicit permission by the provost can override such a restriction. The overall policy can thus be expressed as $o(P_{tutors}\&P_{dept}, P_{provost}, \hat{}[\texttt{blacklisted}(s)])$ whose graphical representation is illustrated in Figure 2(b).

5. PROPERTIES

The formal semantics on which the algebra is based allows us to reason about policy specifications and their properties, meaning correctness requirements that the resulting access control process should satisfy. For instance, consider the hospital policy above, some examples of correctness statements that need to be guaranteed may be:

1. *patient awareness and hospital authorization:* nobody can access `lab_tests` data if there are not both the patient consent and the hospital authorization for it.
2. *law enforcement:* nobody can access new cancer treatments without the approval of the `CCTO`.
3. *explicit denials obedience:* if the oncology department has stated that an access should not be allowed, the access will not be.

These correctness criteria must be satisfied regardless of the contents of the component policies. For instance, property 1 above must be satisfied regardless of which patients gave consents, what accesses the medical department wishes to permit, or how the hospital policy is formulated. Correctness criteria like the ones above can be easily proved exploiting the set theoretic semantics of the algebra. Intuitively, the correctness statements establish that given a template, whatever the structure or content of the specific policies in it, certain conditions are satisfied. The proofs use the formal semantics of expressions to determine whether or not the conditions will be satisfied. In particular, Propositions 5.1 through 5.3 below ensure us, by a simple analysis on the template, that the policy in Figure 2(a) satisfies properties 1, 2, and 3 above.

PROPOSITION 5.1. *Let $T = \tau(X\ Y).o(X, Y, \hat{}c)$ be an incomplete policy expression. For all policy expressions P_1 and P_2, environments e, and authorizations (s, o, a) satisfying c, $(s, o, a) \in [\![T(P_1, P_2)]\!]_e$ if and only if $(s, o, a) \in [\![P_1]\!]_e$ and $(s, o, a) \in [\![P_2]\!]_e$.*

PROOF. By definition, $[\![T(P_1, P_2)]\!]_e = ([\![P_1]\!]_e \setminus [\![P_1\hat{}c]\!]_e) \cup ([\![P_2]\!]_e \cap [\![P_1\hat{}c]\!]_e)$. Since (s, o, a) satisfies c (by hypothesis), we have $(s, o, a) \notin [\![P_1]\!]_e \setminus [\![P_1\hat{}c]\!]_e$, either because $(s, o, a) \notin [\![P_1]\!]_e$, or because $(s, o, a) \in [\![P_1\hat{}c]\!]_e$. It follows that $(s, o, a) \in T(P_1, P_2)$ iff $(s, o, a) \in [\![P_2]\!]_e \cap [\![P_1\hat{}c]\!]_e$; in turn, since (s, o, a) satisfies c, this is equivalent to say that $(s, o, a) \in [\![P_2]\!]_e$ and $(s, o, a) \in [\![P_1]\!]_e$. □

PROPOSITION 5.2. *Let $T = \tau(X\ Y\ Z).(X\hat{}c_1 + (Y\hat{}c_2\&Z))$ be an incomplete policy expression such that $c_1 \wedge c_2$ is not satisfiable. For all policy expressions P_1, P_2, P_3, environments e, and authorizations (s, o, a) satisfying c_2, $(s, o, a) \in [\![T(P_1, P_2, P_3)]\!]_e$ only if $(s, o, a) \in [\![P_3]\!]_e$.*

PROOF. By def., $[\![T(P_1, P_2, P_3)]\!]_e = [\![P_1\hat{}c_1]\!]_e \cup ([\![P_2\hat{}c_2]\!]_e \cap [\![P_3]\!]_e)$. Suppose that (s, o, a) satisfies c_2 and $(s, o, a) \in [\![T(P_1, P_2, P_3)]\!]_e$. Then, $(s, o, a) \notin [\![P_1\hat{}c_1]\!]_e$, otherwise (s, o, a) would satisfy also c_1 contradicting the hypothesis. It follows that $(s, o, a) \in [\![P_2\hat{}c_2]\!]_e \cap [\![P_3]\!]_e \subseteq [\![P_3]\!]_e$. □

PROPOSITION 5.3. *Let $T = \tau(X\ Y\ Z\ W).(X\hat{}c_1 + (Y - Z\hat{}c_2) + W\hat{}c_3)$ be an incomplete policy expression such that $c_1 \wedge c_2$ and $c_2 \wedge c_3$ are not satisfiable. For all policy expressions P_1, P_2, P_3, P_4, environments e, and authorizations (s, o, a) satisfying c_2, if $(s, o, a) \in [\![P_3]\!]_e$, then $(s, o, a) \notin [\![T(P_1, P_2, P_3, P_4)]\!]_e$.*

PROOF. By definition, $[\![T(P_1, P_2, P_3, P_4)]\!]_e = [\![P_1\hat{}c_1]\!]_e \cup ([\![P_2]\!]_e \setminus [\![P_3\hat{}c_2]\!]_e) \cup [\![P_4\hat{}c_3]\!]_e$. Suppose (s, o, a) satisfies c_2. Then, by hypothesis $(s, o, a) \notin [\![P_1\hat{}c_1]\!]_e \cup [\![P_4\hat{}c_3]\!]_e$. Moreover, if $(s, o, a) \in [\![P_3]\!]_e$, then by definition $(s, o, a) \notin ([\![P_2]\!]_e \setminus [\![P_3\hat{}c_2]\!]_e)$. It follows that $(s, o, a) \notin [\![T(P_1, P_2, P_3, P_4)]\!]_e$. □

Note that had the scoping restriction attached to $P_{register}$ not been there, it would have not been possible to prove Proposition 5.3 and, indeed, property 3 would have not been satisfied anymore as stated by the following proposition.

PROPOSITION 5.4. *Let $T = \tau(X\ Y\ Z).(X + (Y\hat{}c\&Z))$ be an incomplete policy expression. There exist policy expressions P_1, P_2, P_3, an environment e, and an authorization (s, o, a) such that $(s, o, a) \in [\![T(P_1, P_2, P_3)]\!]_e$ and $(s, o, a) \notin [\![P_3]\!]_e$.*

PROOF. Take three distinct policy identifiers as P_1, P_2, and P_3, and define $e(P_1) = \{(s, o, a)\}$, and $e(P_2) = e(P_3) = \emptyset$. By definition, $[\![T(P_1, P_2, P_3)]\!]_e = [\![P_1]\!]_e \cup [\![P_2 \hat{} c \& P_3]\!]_e \supseteq [\![P_1]\!]_e = \{(s, o, a)\}$. Then, clearly, $(s, o, a) \in [\![T(P_1, P_2, P_3)]\!]_e$ and $(s, o, a) \notin P_3$. □

In our framework policies can also be analyzed to point out inherent inconsistencies. For instance, a bad formulation of a policy can always cause the result to be *empty*, whatever the structure or contents of its components. This is, for example, the case of a policy of the form $\tau(X\ Y).o(X\&Y, X - Y, Y)$, as stated by the following proposition.

PROPOSITION 5.5. *Let $T = \tau(X\ Y).o(X\&Y, X-Y, Y)$ be an incomplete policy expression. For all policy expressions P_1, P_2, and environments e, $[\![T(P_1, P_2)]\!]_e$ is an empty set.*

PROOF. By definition $[\![T(P_1, P_2)]\!]_e = (([\![P_1]\!]_e \cap [\![P_2]\!]_e) \setminus [\![P_2]\!]_e) \cup (([\![P_1]\!]_e \setminus [\![P_2]\!]_e) \cap [\![P_2]\!]_e)$. First note that $([\![P_1]\!]_e \cap [\![P_2]\!]_e) \setminus [\![P_2]\!]_e$ is empty, as no authorization can be in $[\![P_1]\!]_e \cap [\![P_2]\!]_e$ without being also in $[\![P_2]\!]_e$. Second, note that $([\![P_1]\!]_e \setminus [\![P_2]\!]_e) \cap [\![P_2]\!]_e$ is empty, as no authorization can be in $[\![P_1]\!]_e \setminus [\![P_2]\!]_e$ and $[\![P_2]\!]_e$ at the same time. The proposition immediately follows. □

Note that while the proofs are straightforward (and this is the beauty of the framework), the arguments are not. As authorization languages get more expressive and complete, it is not easy to ensure correctness. Think, for example, of recent logic-based authorization languages, where insertion of a rule or fact can cause an authorization to be derived without the security administrator be aware of it. Being able to state and prove correctness requirements in a very simple way is therefore a great advantage. The simplicity of the correctness statements and proofs is also due to the component-based view supported by the algebra, which can be exploited to reason at different levels of abstraction, considering only the relevant details.

6. EVALUATING POLICY EXPRESSIONS

The resolution of the expression defining a policy P determines a set of ground authorization terms corresponding to the accesses allowed by P. Different strategies can be used to evaluate expressions for enforcing access control. A possible strategy consists in completely resolving the expression (i.e., compiling the policy) and materializing the result as the set of allowed triples. The materialization, against which access control can be efficiently evaluated, will only need to be updated upon changes to the policy.[2] An alternative strategy consists in enforcing a run-time evaluation of each request (access triple) against the policy expression to determine whether the triple belongs to the result (i.e., whether the access should be allowed). Although this does not bear any cost for the complete resolution, it clearly makes the (much more frequent) process of controlling access requests more expensive. Between these two extremes, possibly combining the advantages of them, there are partial evaluation approaches, which can enforce different degrees of computation/materialization. Partial evaluation is particularly appealing as it can combine the advantages of the two solutions: relatively static and known policies can be precomputed and materialized, while more dynamic or unknown policies (like `CCTO` and provost permissions in Section 4) can be evaluated at run-time.

[2]Incremental approaches can be applied to minimize the recomputation of the policy [17].

Before describing access control in more details we illustrate a translation of algebraic expressions into equivalent logic programs, then used for access control enforcement. The main reason for using a logic-based approach is that logic programs provide executable specifications compatible with different evaluation strategies (e.g., Datalog bottom-up engines [11], Prolog top-down evaluation [11], XSB delayed evaluation and tabling [14], Hermes [17]). In particular, partial evaluation techniques permit compilation of the static parts of the policies, thereby improving efficiency. Formal results on logic programs guarantee that the partial evaluation steps preserve the program semantics on the authorization predicates being compiled.

6.1 Translating algebra expressions into logic programs

We present a translation *pe2lp* from policy expressions into logic programs. *pe2lp* creates a distinct predicate symbol for each policy identifier and for each internal node in the syntax tree of the given algebraic expression. For this purpose, a means is needed to denote each operator occurrence (corresponding to different internal nodes); this is accomplished by labeling each such occurrence with a distinct integer, as in $P +_3 Q\&_5 S -_2 R$. Formally, such extended expressions will be called *labeled policy expressions*.

DEFINITION 6.1 (CANONICAL LABELING). *The* canonical labeling *of a policy expression E is the labeled policy expression obtained by numbering the operators in E from left to right with contiguous integers, starting from* 0.

DEFINITION 6.2 (MAIN LABEL). *The* main label *of a labeled policy expression E is the label of the outermost operator of E, that is, the label of the root of the syntax tree of E. If E is simply a policy identifier, then the main label of E is E itself.*

For instance, the canonical labeling of $P + o(Q \hat{} c, S, R)$ is $P +_0 o_1(Q \hat{}_2 c, S, R)$. The main label of this formula is 0. Roughly speaking, the main label of E corresponds to the last operator evaluated, and hence to the "output" of E.

Translation *pe2lp* takes a labeled expression and an environment as input, and returns a logic program "equivalent to" the given expression, in a sense that will be formally specified later on. For each policy identifier P, a predicate $\texttt{auth}_P$ is defined. For each labeled operator op_i, a predicate $\texttt{auth}_i$ is created. All these predicates have three arguments, a subject, an object, and an action. The translation, for an expression E and an environment e, is recursively defined by the table in Figure 3, where P ranges over policy identifiers, and F, G, R range over policy expressions. By $mainp_F$ we shall denote the predicate $\texttt{auth}_\ell$, where ℓ is the main label of F.

DEFINITION 6.3 (CANONICAL TRANSLATION). *The* canonical translation *of a policy expression E w.r.t. an environment e is pe2lp(E^ℓ, e), where E^ℓ is the canonical labeling of E.*

EXAMPLE 6.1. *Let $E = P + o(Q, S, R)$, and let e_0 be the environment mapping P to $\{(\texttt{s}', \texttt{o}', \texttt{a}'), (\texttt{s}'', \texttt{o}'', \texttt{a}'')\}$. Let*

E	*pe2lp*(E,e)
P	$\{\texttt{auth}_P(s,o,a) \mid (s,o,a) \in e(P)\}$ if $e(P)$ is defined, $\emptyset$ otherwise.
$F +_i G$	$\{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z), \texttt{auth}_i(x,y,z) \leftarrow mainp_G(x,y,z)\} \cup pe2lp(F,e) \cup pe2lp(G,e)$.
$F \&_i G$	$\{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge mainp_G(x,y,z)\} \cup pe2lp(F,e) \cup pe2lp(G,e)$.
$F -_i G$	$\{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge \neg mainp_G(x,y,z)\} \cup pe2lp(F,e) \cup pe2lp(G,e)$.
$F \hat{}_i c$	$\{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge c\} \cup pe2lp(F,e)$
$o_i(F,G,R)$	$\{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z) \wedge \neg mainp_R(x,y,z), \texttt{auth}_i(x,y,z) \leftarrow mainp_G(x,y,z) \wedge mainp_R(x,y,z)\} \cup pe2lp(F,e) \cup pe2lp(G,e) \cup pe2lp(R,e)$.
$F *_i R$	$\{\texttt{auth}_i(s,o,a) \leftarrow \texttt{auth}_i(s_1,o_1,a_1) \wedge .. \wedge \texttt{auth}_i(s_n,o_n,a_n) \mid ((s,o,a) \leftarrow (s_1,o_1,a_1) \wedge \ldots \wedge (s_n,o_n,a_n)) \in R\} \cup \{\texttt{auth}_i(x,y,z) \leftarrow mainp_F(x,y,z)\} \cup pe2lp(F,e)$.
$(\tau_i X.F)(G)$	$\{\texttt{auth}_X(x,y,z) \leftarrow mainp_G(x,y,z)\} \cup pe2lp(F,e) \cup pe2lp(G,e)$.

Figure 3: Translation *pe2lp* : from policy expressions to logic programs

Q, S, R be all undefined in e_0. The canonical translation of E w.r.t. e_0 is:

$$\texttt{auth}_P(\texttt{s}', \texttt{o}', \texttt{a}')$$
$$\texttt{auth}_P(\texttt{s}'', \texttt{o}'', \texttt{a}'')$$
$$\texttt{auth}_0(x,y,z) \leftarrow \texttt{auth}_P(x,y,z)$$
$$\texttt{auth}_0(x,y,z) \leftarrow \texttt{auth}_1(x,y,z)$$
$$\texttt{auth}_1(x,y,z) \leftarrow \texttt{auth}_Q(x,y,z) \wedge \neg\texttt{auth}_R(x,y,z)$$
$$\texttt{auth}_1(x,y,z) \leftarrow \texttt{auth}_S(x,y,z) \wedge \texttt{auth}_R(x,y,z)$$

The above translation works correctly if the formal parameters X of the templates occurring in E are all distinct. Formally, we say that E is *clash-free* if for all distinct subexpressions $\tau X.P$, $\tau Y.Q$ of E, it holds that $X \neq Y$. Note that the template semantics does not depend on names of their formal parameter, so we can always rename them uniformly to avoid name clashes in the translation process, and the following proposition holds.

PROPOSITION 6.1. *For each expression E there exists an equivalent clash-free expression E'.*[3]

The following theorem tells us that *pe2lp* is semantics preserving.

THEOREM 6.1. *For all clash-free expressions E and environments e, it holds that $(s,o,a) \in [\![E]\!]_e$ iff $mainp_E(s,o,a)$ belongs to the least Herbrand model of the canonical translation of E w.r.t. e.*

6.2 Access control enforcement

Before illustrating the use of logic programs to enforce access control, we need to specify how to treat "foreign" policies, that is, policies that may be expressed in a different language or stored at other sites. For each foreign policy, a wrapper should be provided [17], that allows our logic programs to query the policy. This can be done with existing logic-based mediator techniques. For instance, using a HERMES-like syntax [17], we may implement the link to an external policy P as $\texttt{auth}_\texttt{P}(s,o,a) \leftarrow \textbf{in}((s,o,a), \texttt{P} : \texttt{grant}())$. Similarly, if the external policy specifies negative authorizations, we may write $\texttt{auth}_{\texttt{P}^-}(s,o,a) \leftarrow \textbf{in}((s,o,a), \texttt{P} : \texttt{deny}())$. In the following, for each policy expression E and environment e, we denote by $\Pi_{E,e}$ the logic program consisting of the canonical translation $pe2lp(E^\ell, e)$ extended with the above wrapper rules for each foreign policy P.

We are now ready to discuss access control and related techniques for partial and complete materialization of policy expressions. In the discussion, we shall assume a given set of policy identifiers, Pol_{dyn}, containing the identifiers of policies that should not be materialized and the base predicates that cannot be evaluated at materialization time.

Partial materialization is accomplished by applying standard *partial evaluation* [16] techniques to the logic program $\Pi_{E,e}$. We recall that partial evaluation transforms program rules by iteratively applying *unfolding steps* of the form:

$$(A \leftarrow B_1, \ldots, B_n) \Rightarrow (A \leftarrow B_1, \ldots, B_{i-1}, C_1, \ldots, C_m, B_{i+1}, \ldots, B_n)\theta$$

where $H \leftarrow C_1, \ldots, C_m$ is a program clause whose variables are renamed with fresh variables, and θ is the *most general unifier* [11] of B_i and H. The following predicates are not unfolded:

- Predicates $\textbf{in}(\ldots, P : \ldots)$ such that $P \in \mathsf{Pol}_{dyn}$.
- Predicates of the form $\texttt{auth}_P$ such that $P \in \mathsf{Pol}_{dyn}$, unless $\texttt{auth}_P$ is implemented by a wrapper.
- Base predicates in Pol_{dyn}.

Eventually, rule bodies will contain only predicates of this kind, and partial evaluation terminates. By well-known logic programming results, the following proposition holds ([16]):

PROPOSITION 6.2. *Let $\mathsf{PartEv}(\Pi_{E,e})$ be the result of the partial evaluation of $\Pi_{E,e}$, and $mainp_E$ the main predicate of E's canonical translation. Then, for all ground authorizations (s,o,a), $\mathsf{PartEv}(\Pi_{E,e}) \models mainp_E(s,o,a)$ iff $\Pi_{E,e} \models mainp_E(s,o,a)$.*

Intuitively, this proposition says that partial evaluation preserves the meaning of the original logic program. As a corollary of this proposition and Theorem 6.1, the partially evaluated (or *partially materialized* [4]) program $\mathsf{PartEv}(\Pi_{E,e})$ captures exactly the meaning of the policy expression E in environment e.

COROLLARY 1. *For all ground authorizations (s,o,a), $\mathsf{PartEv}(\Pi_{E,e}) \models mainp_E(s,o,a)$ iff $(s,o,a) \in [\![E]\!]_e$.*

EXAMPLE 6.2. *Consider the University Laboratory Policy in Figure 2(b). Assume that, at materialization time, the P_{tutors} and P_{dept} policies are known and consist of $\{(s_1,o_1,a_1), \ldots, (s_n,o_n,a_n)\}$ and $\{(s_1,o_1,a_1), \ldots, (s_k,o_k,a_k)\}$ $(k > n)$, respectively; while $P_{provost}$ policy and predicate* `blacklisted` *are unknown, meaning $\mathsf{Pol}_{dyn} = \{P_{provost}, \texttt{blacklisted}\}$. The canonical translation of the policy, w.r.t. the environment e binding the P_{tutor} and P_{dept} to the above triples and leaving policy $P_{provost}$ and predicate* `blacklisted` *undefined, is:*

$$\texttt{auth}_0(x,y,z) \leftarrow \texttt{auth}_1(x,y,z) \wedge \neg\texttt{blacklisted}(x)$$
$$\texttt{auth}_0(x,y,z) \leftarrow \texttt{auth}_{provost}(x,y,z) \wedge \texttt{blacklisted}(x)$$
$$\texttt{auth}_1(x,y,z) \leftarrow \texttt{auth}_{tutors}(x,y,z) \wedge \texttt{auth}_{dept}(x,y,z)$$
$$\texttt{auth}_{tutor}(s_i,o_i,a_i) \quad (i = 1,\ldots,n)$$
$$\texttt{auth}_{dept}(s_i,o_i,a_i) \quad (i = 1,\ldots,k)$$

[3] Two expressions E, F are equivalent iff for all environments e, $[\![E]\!]_e = [\![F]\!]_e$.

[4] Complete materialization is a special case of partial evaluation, where Pol_{dyn} is empty and every predicate is unfolded.

We extend this program with the wrapper rule

$$\texttt{auth}_{provost}(x, y, z) \leftarrow \textbf{in}((x, y, z), P_{provost}\texttt{: grant}()).$$

Then, we partially evaluate this program, obtaining:

$$\texttt{auth}_0(s_i, o_i, a_i) \leftarrow \neg\texttt{blacklisted}(s_i)$$
$$\texttt{auth}_0(s_i, o_i, a_i) \leftarrow \textbf{in}((s_i, o_i, a_i), P_{provost}\texttt{: grant}()) \wedge \texttt{blacklisted}(s_i)$$

where $i = 1, \ldots, n$.
Note that several intermediate predicate calls have been removed from the partially evaluated (or materialized) program.

7. ELEMENTARY POLICY SPECIFICATION

Our algebra can combine policies stated in different languages and through different paradigms. The algebra is therefore not substitutive of authorization languages, but complements them by allowing different specifications to be merged together, and combined according to different options. The independence from and support for the coexistence of different authorization languages and control mechanisms is a considerable advantage of our approach.

We note however, that when no other authorization language is being applied already, the constructs of our algebra can be also used to specify elementary policies.

In particular, the traditional *closed* policy can be expressed as a policy "P" listing the authorization triples corresponding to the accesses to be allowed; while the *open* policy can be expressed as a policy "$P_{All} - P$", where P_{All} is bound to the set of all possible authorizations and P contains the accesses to be denied. Authorizations for user groups and data types that propagate to their members can be simply expressed in our framework through rules that enforce authorization propagation along the hierarchy. The authorization specifications, stated with respect to individual elements (e.g., users or documents) or classes thereof (e.g., user groups or directories), would then be closed by the set of derivation rules $R = \{(s, o, a) \leftarrow (s', o, a), s \leq s', (s, o, a) \leftarrow (s, o', a), o \leq o', (s, o, a) \leftarrow (s, o, a'), a \leq a'\}$, where $\leq$ reflects the order defined on the different dimensions [7]. Derivation of authorizations according to criteria other than hierarchical relationships can be enforced in an analogous way.

Recent authorization models support both positive and negative authorizations. Positive authorizations state permissions whereas negative authorizations state denials [12]. The way positive and negative authorizations interplay is established by overriding/conflict resolution rules that may depend on the hierarchical relationships of the authorization elements and/or on priorities (or types) associated with the authorizations. Although we do not explicitly support negative authorizations, our framework can indeed express denials through the subtraction operator (negative authorizations appear as a policy to be removed). For instance, a *hybrid specification* with a *denial take precedence* policy can be enforced by simple expression "$P^+ - P^-$" where P^+ are the positive authorization terms and P^- are the authorization terms to be negated. Conflict resolution policies based on the hierarchies of the data system can be supported in an analogous way. As an example, the *most specific takes precedence* policy w.r.t. a hierarchy is obtained by computing, for each node the sum of its positive statements minus the sum of the negative statements for its descendants, and summing up all the triples returned for each node. Here, by statements we mean the authorization triples closed under the propagation rules for the considered hierarchy. Formally, let i be the different nodes in the hierarchy w.r.t. which the policy is stated[5], P_i^+ and P_i^- the authorization triples corresponding to permissions and denials referred to i, and R_H the hierarchy-based propagation rules. Then, the most specific takes precedence policy w.r.t. hierarchy H is defined as $\sum_{i \in H}(P_i^+ * R_H - \sum_{j \leq i} P_j^- * R_H)$.[6] Alternatively, the most specific take precedence principle can be achieved by closing the given authorizations under suitable propagation rules enforcing the criteria [8]. As another example, consider the Orion authorization model [13], where authorizations propagate down the hierarchies and are classified as *strong* or *weak*. Strong authorizations (guaranteed to be free of conflict among themselves) override weak authorizations. Conflicts between weak authorizations are solved according to the most specific take precedence policy. The overall policy can be stated as "$P_{weak} + P^+_{strong} * R_H - P^-_{strong} * R_H$", where P^+_{strong} and "$P^-_{strong} * R_H$" are the positive and negative authorizations, respectively, R_H are the hierarchy-based propagation rules, and P_{weak} is the set of triples resulting from applying the most specific takes precedence policy to weak authorizations as stated above.

Approaches enforcing further overriding criteria, for example, inclusion of organizational-level vs site-level authorizations [4], or explicit priorities, such as the order in which authorizations are listed [15], can be expressed in a similar way. Intuitively, in the expressions enforcing authorizations w.r.t. a given criteria, triples denoting permissions appear as policies to be added, while triples denoting denials appear as policies to be subtracted. The order in which policies appear in the expression determines which policy override which.

Our framework is therefore able to support and combine different approaches existing in the literature. In this respect, algebraic expressions turn out to be very flexible: a new dimension/criterion to be taken care of is simply reflected in the introduction of one (or two, if negative authorizations are supported) operands in the expression. This has also advantages in terms of clarity and readability of the specifications.

8. CONCLUDING REMARKS

We presented an algebra for composing access control policies. We here summarize how our approach addresses the different requirements discussed in Section 2.

1. *Heterogeneous policies* can be supported either by exploiting the algebra constructs to represent the different policies (see Section 7) or by referring to heterogeneous policies through policy identifiers then interpreted by means of wrappers (see Section 6.2).

[5]The hierarchy can be the hierarchy of subjects, objects, or actions hierarchy or a combination of them [8].

[6]In case of incomparable conflicts, this expression resolves in favor of positive authorizations. A denial take precedence principle could be enforced by subtracting from the result the sum of the nonoverridden negative statements, obtained with the dual expression $\sum_{i \in H}(P_i^- * R_H - \sum_{j < i} P_j^+ * R_H)$.

2. *Unknown policies* are supported by means of policy identifiers that can remain unbound in the environment. The translation of expressions into logic programs and the application of partial evaluation techniques on them allow us to delay the evaluation of unknown policies until run time, without need of redoing the whole computation, and guarantee the correctness of the resulting controls (see Section 6.2). Furthermore, templates allow the expression of incomplete policies in the formal semantics and the proofs of correctness properties on incomplete specifications (see Section 5).

3. *Interference* of program rules and authorizations coming from different policies is controlled by restricting rule application to specific policies by means of the closure construct.

4. *Expressiveness* is achieved by the different operators that easily allow the formulation of protection restrictions as illustrated in the examples and discussions contained herein.

5. *Different abstraction levels* are naturally supported by the component-based approach. Each component may internally be structured in sub-components and security administrators can zoom in the different policies or look at a higher level view as desired (see Figure 2(a)). Templates provide a formal means to operate on the different levels as one may simply look at the template (higher abstraction level) or at its actual parameters corresponding to the contents (zoom in) of the formal parameters.

6. *Formal semantics* has been provided in Section 3. We have also illustrated how the algebra semantics can be exploited to reason about properties of the specifications and not only to implement them. Our algebra is implementation independent and can be used to design, analyze, and combine requirements in different systems.

This paper is only the first step towards the definition of a formal and flexible access control composition framework and leaves space for further work. Future work to be carried out includes investigation of administrative policies for regulating the specification of the different component policies by different authorities; the analysis of incremental approaches to enforce changes to component policies; application of the evaluation of automated deduction techniques to prove properties of the specifications; and the performance assessment of different partial evaluation techniques.

9. REFERENCES

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages*, 14(4):1–60, October 1992.

[2] D. Banisar and S. Davies. *Privacy & Human Rights - An International Survey of Privacy Laws and Developments*. EPIC, 1999.

[3] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2):101–140, April 1999.

[4] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. In *Proc. of the WWW9 Conference*, Amsterdam, May 2000.

[5] H.H. Hosmer. The multipolicy paradigm. In *Proc. of the 15th National Computer Security Conference*, pages 409–422, Baltimore, MD, October 1992.

[6] T. Jaeger. Access control in configurable systems. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming*, volume 1603, pages 289–310. Springer-Verlag, 1999.

[7] S. Jajodia, P. Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proc. of the 1997 IEEE Symposium on Security and Privacy*, pages 94–107, Oakland, CA, May 1997.

[8] S. Jajodia, P. Samarati, V.S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. of the 1997 ACM SIGMOD*, pages 474–485, Tucson, AZ, May 1997.

[9] C. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[10] N. Li, J. Feigenbaum, and B. Grosof. A logic-based knowledge representation for authorization with delegation. In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, pages 162–174, Mordano, Italy, June 1999.

[11] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1984.

[12] T.F. Lunt. Access control policies for database systems. In C.E. Landwehr, editor, *Database Security II: Status and Prospects*, pages 41–52. North-Holland, 1989.

[13] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM TODS*, 16(1):89–131, March 1991.

[14] K. Sagonas, T. Swift, D.S. Warren, J. Freire, and P. Rao. The XSB programmer's manual, version 2.2. http://xsb.sourceforge.net, April 2000.

[15] H. Shen and P. Dewan. Access control for collaborative environments. In *Proc. Int. Conf. on Computer Supported Cooperative Work*, pages 51–58, Toronto, Canada, November 1992.

[16] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, Cambridge, MA, 1997.

[17] V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J.J. Lu, A. Rajput, T.J. Rogers, R. Ross, and C. Ward. Hermes: Heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/publications/abstracts/hermes.html.

[18] T.Y.C. Woo and S.S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2,3):107–136, 1993.