Modular Access Control via Strategic Rewriting

Daniel J. Dougherty[1], Claude Kirchner[2], Hélène Kirchner[3], Anderson Santana de Oliveira[2]

[1] Worcester Polytechnic Institute
[2] INRIA & LORIA[⋆]
[3] CNRS & LORIA

Abstract. Security policies, in particular access control, are fundamental elements of computer security. We address the problem of authoring and analyzing policies in a modular way using techniques developed in the field of term rewriting, focusing especially on the use of rewriting strategies. Term rewriting supports a formalization of access control with a clear declarative semantics based on equational logic and an operational semantics guided by strategies. Well-established term rewriting techniques allow us to check properties of policies such as completeness and the absence of conflicts. A rich language for expressing rewriting strategies is used to define a theory of modular construction of policies in which we can better understand the preservation of properties of policies under composition. The robustness of the approach is illustrated on the composition operators of XACML.

1 Introduction

Access control is at the heart of computer security. It has grown beyond mediating operating-system interactions between users and files and now plays a central role in web-based systems, legal policies, and business rules. Accompanying these expanded applications of access control, our conception of the mechanism of authorization now goes beyond the classical model [29] of access-control matrices, and we now view access control decisions as the embodiment of a set of rules. We call such a set of rules an access-control *policy*. Although monitoring and enforcement mechanisms are important aspects of the study of access control, the size and complexity of the systems being treated mean that the policies themselves are interesting software artifacts in their own right. They are sensitive to complex conditions on the policy environment, which represents the data that a program respecting the policy manipulates, such as attributes of subjects and resources and relations among these. They are not easy to get right.

In light of these considerations it is now typical in large or complex systems to disentangle policy from application code. They are written in domain-specific, typically declarative languages, and reasoning about the correctness of policies is a subtle matter. It is common wisdom that a key to designing, reasoning about, and maintaining a large system is modularity, with corresponding attention to the mechanisms by which the models in a system interact.

[⋆] LORIA: UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP; Nancy, France.

In this paper we are interested in the question of building access-control policies in a modular fashion, and taking some initial steps towards a theory of how parts of a policy interact.

We propose *term rewriting* [39, 2] as a formalism for representing access control policies. Rewriting is a well-established paradigm whose applications include theoretical foundations for functional programming languages and theorem provers. It is flexible and expressive enough to capture a wide range of policy framework arising in practice and indeed it is a universal model of computation. It has a clean declarative semantics, based on equational logic. There is an active research community supporting efficient implementations and tools for reasoning about properties such as termination and confluence of systems. One can view rewrite systems as an intermediate language for policies; our thesis in this paper is that some of the more interesting aspects of reasoning about policies are profitably viewed in this context.

Indeed, term rewriting is not a single formalism but rather a family of variations on a robust paradigm of directed equality. It is easy to see that simple term rewriting can capture polices such as Unix file-permissions rules, the richer setting of conditional rewriting is as rich as the language of Datalog explored by several authors (notably in trust-management research), and—as sketched below—core XACML polices can be captured by rewriting modulo associativity-commutativity.

To give a flavor of how term rewriting can capture policy rules we may consider the following rules, adapted from the XACML specification [36]:

– A person, identified by his or her social security number, may read any record for which he or she is the designated patient:

$$req(patient(x), read, record(x)) \rightarrow permit.$$

Here *patient* names the function from patient numbers to patients as Subjects and *record* is a function from patient numbers to health records as Resources, while *read* is a constant symbol, of the sort Actions.
The variable x is implicitly universally quantified, so that the rewriting above captures the generality of the authorization rule; and the repetition of the variable as a parameter has the effect of enforcing the binding between the patient and her record.
– An administrator shall not be permitted to write to medical elements of a patient record.

$$req(admin(x), write, record(y)) \rightarrow deny$$

Here *any* administrator, as named perhaps by his/her employee number is denied write access to *any* health record: note the use of distinct variables in the rule. Also note the use of explicit *deny* as a decision. It is crucially important to modularity of policies that *deny* is not treated as the negation of *permit*: this will be further illustrated in the body of the paper.
– (Inheritance of authority) It is straightforward to capture certain notions of authorization hierarchy. For example, to say that subject s_2 inherits from subject s_1 all access rights involving resource r, it suffices to have the following rule in a policy.

$$req(s_1, x, r) \rightarrow req(s_2, x, r)$$

Here x is a variable ranging over actions. Note that this rule is a refinement of the type of inheritance typically incorporated into a Role-based Access Control Model (in which one role may inherit all privileges from another, uniformly across all actions and resources).

In a large organization, there are many classes of "Subjects" with different needs for access to an immense variety of "Resources". For example, in a hospital there will be rules governing the access of patients to their health records, their financial records, and the like, while at the same time, there will be rules for employee access to these same records as well as to resources quite different from health records. Meanwhile, other entities such as insurance carriers will be subject to yet another set of rules for access to this data and more.

The different constituencies (patients, staff, insurers) are almost certainly going to have somewhat different — even competing— requirements on their use of the data and place different emphases on the security goals (confidentiality, availability, integrity) of policies. It is natural to imagine that the sets of rules describing these various modes of access should not be authored and maintained in a single monolithic policy. In this setting, *the theory of composition* of policies becomes crucially important.

As a very simple example, imagine that rules for patient data access and rules for staff data access are composed in separate policy documents, $\wp_p$ and $\wp_s$ respectively. What should we say about the decision of $\wp_p$ in the context of a request by an administrator to write a health record? Assuming $\wp_p$ will not explicitly compute a decision (permit or deny) upon such a request, we must uniformly assume a *default* decision, perhaps default-deny, for all requests not handled directly. But this immediately leads to the conclusion that composing policies is something more subtle than taking their union. Consider by contrast a request by an administrator to read the next-of-kin information for a patient. A default-deny by $\wp_p$ for this request would mean that when $\wp_p$ was combined with $\wp_s$, which may explicitly compute a *permit* for this request, the resulting logical theory, taken in a naive sense, would be contradictory.

So at the very least one must make a distinction between a policy decision which is computed in a "direct" way from the policy rules, and one taken as a default. The situation is even more interesting if two modules of a policy compute contradictory decisions: if the policy is to be coherent in practice there must be a principled way to combine the modules, a mechanism that lends itself to clean design and supports analysis and verification. The combination method we explore in this paper is that of *rewriting strategies*.

The need for flexibility is addressed by the design of recent proposals of specification languages for access control [12, 21, 36]. These languages associate a rule-based formalism with partial policy specifications, that assume not only positive and negative authorization rules, but dispose of a larger set of possible decisions, such as *permit, deny, not applicable,. . .*

The remaining sections of this paper are organized as follows: we recall in Section 2 the main notions on rewrite rules and strategies used in this paper. In Section 3 we give the definition, suitable properties and examples of an access control policies expressed in the rewrite-based framework. We formalize policy composition in Section 4, as well as suitable properties of policy composition and we illustrate our approach in particular on the composition operators of XACML. We discuss related and further works in Section 5.

2 Background

Basic definitions on term rewriting can be found in [39, 2]. Let us recall those which are used in the following. A many-sorted signature $(\mathcal{S}, \mathcal{F})$, or $\mathcal{F}$ for short, is a set of sorts $\mathcal{S}$ and a set of function symbols $\mathcal{F}$. Each $f \in \mathcal{F}$ has a profile $f : S_1 \times \ldots \times S_n \rightarrow S$, where $S_1, \ldots S_n, S \in \mathcal{S}$, and is associated to a natural number by the arity function (ar : $\mathcal{F} \rightarrow \mathbb{N}$). When $ar(f) = 0$, the function symbol f is called a constant.

$\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of well-sorted terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. For $f \in \mathcal{F}$, $f(\mathcal{T}(\mathcal{F}), \ldots, \mathcal{T}(\mathcal{F}))$ denotes the set of ground terms with f as top symbol.

A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, with a finite domain $\{x_1, \ldots, x_k\}$ and written $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$.

A rewrite rule is an ordered pair of terms, denoted as $l \rightarrow r$, $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, where l is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ such that l and r belong to the same sort. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A rewrite system is a (finite or infinite) set of rewrite rules. Rules can be labeled to easily distinguish among them. A rewrite rule $l \rightarrow r$ is a *collapsing rule* if r is a variable. It is a *duplicating rule* if there exists a variable that has more occurrences in r than in l.

Given a rewrite system R on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, a function symbol which is not the top symbol of any rule in R is called a *constructor*. Others symbols are called *defined functions*. A *constructor system* $(\mathcal{C}, \mathcal{D}, R)$ is defined by a set of constructors $\mathcal{C}$, a set of defined functions $\mathcal{D}$ and a set of rewrite rules R, such that every left-hand side of any rule in R is of the form $f(t_1, ..., t_n)$ with $f \in \mathcal{D}$ and $t_1, ..., t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Two constructor systems $(\mathcal{C}_1, \mathcal{D}_1, R_1)$ and $(\mathcal{C}_2, \mathcal{D}_2, R_2)$ share constructors if $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{C}_1 \cup \mathcal{C}_2$ are pairwise disjoint.

Given a rewrite system R, a term t rewrites to a term t', which is denoted $t \rightarrow_R t'$ if there exists a rewrite rule $l \rightarrow r$ of R, a position ω in t, a substitution σ, satisfying $t_{|\omega} = \sigma(l)$, such that $t' = t[\sigma(r)]_\omega$.

A rewriting derivation of the rewrite system R is any sequence of rewriting steps $t_1 \rightarrow_R t_2 \rightarrow_R \ldots$. The *source* of such a derivation is t_1. When the derivation is finite, its last term is called its *target*. R induces a derivability relation $\xrightarrow{*}_R$ on terms: $t \xrightarrow{*}_R t'$ if there exists a rewriting derivation from t to t'. If the derivation contains at least one step, it is denoted by $\xrightarrow{+}_R$. A rewrite system is terminating (or strongly normalizing) if all rewriting derivations are finite. A term t is R-normalized (or in R-*normal form*) when the empty derivation is the only one with source t; a derivation is *normalizing* when its target is R-normalized. A rewrite system R is *weakly terminating* if every term t is the source of a normalizing derivation. It is confluent if for all terms t, u ,v, $t \xrightarrow{*}_R u$ and $t \xrightarrow{*}_R s$ implies $u \xrightarrow{*}_R s$ and $v \xrightarrow{*}_R s$, for some s. When it is clear from the context, we may omit the index R.

The notion of strategy is fundamental in general as well as in this paper, and we give here a general presentation of the main ideas. We use a general definition, slightly different from the one used in [39]: a *rewrite strategy* ζ for the rewrite system R is a subset of the set of all derivations of R. The *application of a strategy* ζ on a term t is denoted $[\zeta](t)$

and defined as the set of all targets t' of the derivations of source t in ζ. The *domain* of a strategy is the set of terms that are source of a derivation in ζ. When no derivation in ζ has for source t, we say that the strategy application on t fails. The result of the application of a failing strategy on a term t is the empty set. In this paper, we will consider only strategies that are stable by concatenation (i.e. $t \xrightarrow{*}_R t' \in \zeta$ and $t' \xrightarrow{*}_R t'' \in \zeta$ implies $t \xrightarrow{*}_R t' \xrightarrow{*}_R t'' \in \zeta$). Note that the rewrite rules in R can be considered as elementary or atomic strategies.

For instance, if a and b are constants, the application of the rewrite rule $a \to b$ to the term a is denoted $[a \to b](a)$ and evaluates to $\{b\}$.

A strategy could be described by enumerating all its elements or more suitably by a *strategy language*. From elementary strategies expressions directly issued from a rewrite system R, more elaborated strategies expressions are built like in ELAN [25], Stratego [42], Tom [3] or more recently MAUDE [32]. The semantics of such a language is naturally described in the rewriting calculus [9, 10]. We describe below the main elements of the strategy language of interest in this paper. Most of them are available in Tom [35][4].

Given a rewrite system R over $\mathcal{T}(\mathcal{F}, \mathcal{X})$, a strategy expression is either a rewrite rule in R or an expression described below. A strategy expression ζ may take arguments $\zeta_1, \ldots, \zeta_n$, and the resulting expression is expressed functionally: $\zeta(\zeta_1, \ldots, \zeta_n)$. Notice that this is consistent with the notation $\zeta(R)$ as soon as the definition of ζ does not depend on is arguments order. When it is clear from the context, we identify the strategy expression and the strategy (i.e. the set of derivations it represents). In a consistent way, the application of a strategy expression to a term is defined as the application of the strategy it represents.

A simple strategy is the sequential application of two rules. It is described by the concatenation operator "seq". For instance $[\texttt{seq}(l_1 \to r_1, l_2 \to r_2)](t)$ denotes $[l_2 \to r_2]([l_1 \to r_1](t))$. This strategy operator extends naturally to multiple arguments:

$$[\texttt{seq}(\zeta_1, \ldots, \zeta_n)](t) \quad = \quad [\zeta_n]([\zeta_{n-1}](\ldots [\zeta_1](t)))$$

Identity and failure are strategies easy to imagine:

$$\begin{array}{lcl} [\texttt{id}](t) & = & \{t\} \\ [\texttt{fail}](t) & = & \emptyset \end{array}$$

The strategy computing all derivations issued from the application of a rewrite system R is called `universal`; it takes as argument the set of rules under consideration:

$$[\texttt{universal}(R)](t) \quad = \quad \{t' \mid t \xrightarrow{*}_R t'\}$$

For instance, we have:

$$\begin{array}{lcl} [\texttt{universal}(a \to a)](a) & = & \{a\} \\ [\texttt{universal}(f(x) \to f(f(x)))](f(a)) & = & \{f(a), f(f(a)), f(f(f(a))), \ldots\} \end{array}$$

[4] http://tom.loria.fr

One can successively try to apply several strategies using the `choice` operator (which corresponds to *first* in ELAN): its first argument is applied if it does not fail, otherwise the second one is applied (and may fail too).

$$\begin{array}{lcll} [\texttt{choice}(\zeta_1,\zeta_2)](t) & = & [\zeta_1](t) & \text{if } [\zeta_1](t) \neq \emptyset \\ [\texttt{choice}(\zeta_1,\zeta_2)](t) & = & [\zeta_2](t) & \text{if } [\zeta_1](t) = \emptyset \end{array}$$

Clearly `choice` is associative and therefore its syntax is extended to be applicable to a list of strategies:

$$\texttt{choice}(\zeta_1,\zeta_2,\ldots,\zeta_n) \;=\; \texttt{choice}(\zeta_1,\texttt{choice}(\zeta_2,\ldots,\zeta_n))$$

Other strategies allow controlling the application of rules over sub-terms of a term. The strategy `one` must succeed on at least one of the sub-terms of a term. On the other hand, `all` application must succeed on each sub-term, otherwise, the result is failure:

$$\begin{array}{lcl} [\texttt{one}(\zeta)](f(t_1,\ldots,t_n)) & = & f(t_1,\ldots,[\zeta](t_i),\ldots,t_n),\ \text{if } [\zeta](t_i) \neq \emptyset \\ [\texttt{all}(\zeta)](f(t_1,\ldots,t_n)) & = & f([\zeta](t_1),\ldots,[\zeta](t_n)),\ \text{if } \forall i \in \{1,\ldots,n\}, [\zeta](t_i) \neq \emptyset \end{array}$$

Using the above set of operators, we can define recursive ones which iterate the application of a strategy to a term, for example:

$$\begin{array}{lcl} \texttt{try}(\zeta) & = & \texttt{choice}(\zeta,\texttt{id}) \\ \texttt{repeat}(\zeta) & = & \texttt{try}(\texttt{seq}(\zeta,\texttt{repeat}(\zeta))) \end{array}$$

It is worth noticing that `try` and `repeat` never fail. Other high level strategies implement term traversal and normalization on terms and are well-known in the rewrite system literature:

$$\begin{array}{lcl} \texttt{topDown}(\zeta) & = & \texttt{seq}(\zeta,\texttt{all}(\texttt{topDown}(\zeta))) \\ \texttt{bottomUp}(\zeta) & = & \texttt{seq}(\texttt{all}(\texttt{bottomUp}(\zeta)),\zeta) \\ \texttt{OnceTopDown}(\zeta) & = & \texttt{choice}(\zeta,\texttt{one}(\texttt{OnceTopDown}(\zeta))) \\ \texttt{OnceBottomUp}(\zeta) & = & \texttt{choice}(\texttt{one}(\texttt{OnceBottomUp}(\zeta)),\zeta) \\ \texttt{innermost}(\zeta) & = & \texttt{repeat}(\texttt{onceBottomUp}(\zeta)) \\ \texttt{outermost}(\zeta) & = & \texttt{repeat}(\texttt{onceTopDown}(\zeta)) \end{array}$$

Example 1. Some examples of strategy application are:

$$\begin{array}{lcl} [\texttt{universal}(a{\to}b, a{\to}c)](a) & = & \{a,b,c\} \\ [\texttt{choice}(a{\to}b, a{\to}c)](a) & = & \{b\} \\ [\texttt{choice}(a{\to}c, a{\to}b)](b) & = & \emptyset \\ [\texttt{try}(b{\to}c)](a) & = & \{a\} \\ [\texttt{repeat}(\texttt{choice}(b{\to}c, a{\to}b))](a) & = & \{c\} \end{array}$$

3 Rewrite-Based Policies

Classically, access control concerns establishing which actions are allowed to be executed by the active entities of a system (e.g. users, processes, roles, etc), called *principals* or *subjects*, over its protected entities (files, databases, printers, etc), called *resources* or *objects* [15]. Recent developments are aimed to express various constraints on the environment where policies run, in order to capture real world requirements from policy authors, such as time, location, and any other condition involving attributes of principals and objects.

In this context, it is important to embark expressive computational power in the definition of policies. As the notion of pattern and of rule is quite natural in the context of policies specifications, we propose here a quite general definition of access control, based on the full power of strategic rewriting.

In our model, authorization decisions are computed by a set of rewrite rules that transform the input terms, representing access requests, into authorization terms. In order to take the raw computational power of term rewriting and to enhance the agility of the policy specification language, we use strategies to explicitly control the rules application. We define rewrite-based policies as follows, where Q stands for queries (or requests) and D for decisions.

Definition 1 (Security Policy). *A access control security policy, $\wp$, is a 5-tuple $(\mathcal{F}, D, R, Q, \zeta)$ such that:*

1. *$\mathcal{F}$ is a signature;*
2. *D is a non-empty set of ground terms: $D \subseteq \mathcal{T}(\mathcal{F})$;*
3. *R is a set of rewrite rules over $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
4. *Q is a set of terms from $\mathcal{T}(\mathcal{F})$: $Q \subseteq \mathcal{T}(\mathcal{F})$;*
5. *ζ is a rewrite strategy for R.*

Let us explain the main design choices made in this definition.

– First we consider that the policy specification and its environment are described as terms built over the signature $\mathcal{F}$. The set of possible decisions to be taken by the policy is denoted by D. Indeed, D is often a set of constants and the two main constants in D are usually $permit$ and $deny$. But since it is crucial to model also policies that do not directly take decision, it can be useful to have a constant $not\ applicable$ that simply expresses the fact that the current policy in the current context cannot decide about the access. Moreover, the result returned by a policy could be more elaborated than just a constant and can be a ground term containing further information. Whatever the set D contains, we assume it to be non-empty. What is significant is not treating the failure to derive a permission as a denial. In contrast to [21], in which this later design is followed, we can treat explicitly decisions such as $deny$ and $not\ applicable$. This is a crucial advantage for merging rules, since in purely logic-based works, there is no way to handle in the theory what happens when a policy which derives $deny$ for a request q is merged with another which then derives $permit$ explicitly, for the same q.
– The rewrite system R describes the behavior of the policy as well as some necessary computations which explain how its environment evolves. The role of the strategy is to point derivations of R whose interest is to produce decisions.
– The requests are a subset of ground terms. They typically express questions of the form: is a certain entity authorized to access a resource given the current configuration of the policy environment.

– The last component is the strategy which allows one to finely specify the evaluation order of the policy rules.

One of the main nice consequences of this approach, in addition to its expressivity, which we illustrate on the examples below, is that it allows us to take advantage of all the results obtained by the rewriting community since the last thirty years. Amongst such results, we investigate confluence and termination.

Example 2. A simple example, inspired from [5], illustrates the above definition by assigning authorizations based on a "user id" which is represented by a natural number: all requests from user whose "id" is bigger than three are denied.

– Let the policy signature be: $\mathcal{F} = \{0 : Nat, s : Nat \rightarrow Nat, + : Nat \times Nat \rightarrow Nat, auth : Nat \rightarrow A, permit : A, na : A, deny : A\}$
– The set of of constant symbols representing decisions is $D = \{permit, na, deny\}$
– Consider R as the following set of rules (the operator s gives the successor of a number, $+$ is the usual sum operator, x, y are variables of sort Nat:):

$$\begin{array}{ll} x + s(y) & \rightarrow s(x+y) \\ x + 0 & \rightarrow x \\ auth(0) & \rightarrow permit \\ auth(s(0)) & \rightarrow permit \\ auth(s(s(0))) & \rightarrow na \\ auth(s(s(s(x)))) & \rightarrow deny \end{array}$$

– the set Q contains ground terms with top symbol $auth$;
– A possible strategy for this policy, among others that guarantee a normalization process, is $\zeta = \texttt{innermost}(R)$.

This defines a security police as all conditions of Definition 1 are satisfied. An example of request evaluation is: $[\zeta](auth(s(0) + s(s(s(0))))) = \{deny\}$

Example 3. As already suggested in the introduction, we can model a policy for a clinical system (this example is adapted from the XACML specification [36], and first presented in the rewrite-based formalism in [12]).

– The policy signature, $\mathcal{F}$, contains the following symbols:

$$\begin{array}{lllll} auth & : & Request \times Condition & \rightarrow & A \\ req & : & Subject \times Action \times Object & \rightarrow & Request \\ read,\ write & : & & \rightarrow & Action \\ permit,\ deny,\ na & : & & \rightarrow & A \\ patient,\ phy & : & Number & \rightarrow & Subject \\ admin,\ per & : & Number & \rightarrow & Subject \\ record & : & Number & \rightarrow & Object \\ guard & : & Subject \times Subject & \rightarrow & Condition, \\ respPhy & : & Subject \times Subject & \rightarrow & Condition \\ urgency & : & & \rightarrow & Condition \end{array}$$

- The set of decisions is $D = \{permit, deny, na\}$.
- R is the following set of rules, where variables are $x, y : Number$; $r : Object$; $c : Condition$:

$$\begin{array}{ll} auth(req(patient(x), read, record(x)), c) & \rightarrow permit \\ auth(req(per(x), read, record(y)), guard(per(x), patient(y)))) & \rightarrow permit \\ auth(req(phy(x), read, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\ auth(req(phy(x), write, record(y)), respPhy(phy(x), patient(y))) & \rightarrow permit \\ auth(req(admin(x), read, r), c) & \rightarrow deny \\ auth(req(admin(x), write, r), c) & \rightarrow deny. \end{array}$$

  In the order of appearance these rules state that: a patient can read his own record, the guardian of a person can read the record for that person, the responsible physician of a patient can read or write data for her record, the last two rules deny any access of administrators to records.
- The set of requests, Q, is the set of all terms of the form $auth(\mathcal{T}(\mathcal{F}), \mathcal{T}(\mathcal{F}))$.
- One could adopt the strategy $\zeta = \texttt{choice}(R, auth(q, c) {\rightarrow} na)$, which introduces a default rule for this policy, where $q : Request$. The terms in Q which are not reduced by the rules from R will be rewritten into na, which ensures completeness. The example presented here is a security policy according to Definition 1.

The policy illustrated in this example has some desirable properties; for example the evaluation of a request is guaranteed to return a unique result, as will be demonstrated shortly.

A security policy is consistent if it computes at most one authorization decision:

Definition 2 (Consistency). *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is consistent if for every query $q \in Q$, ζ applied to q returns at most one result: $\forall q \in Q$, the cardinality of $[\zeta](q)$ is less than or equal to* 1.

This means that for every query evaluation, a deterministic result is computed by the application of ζ on the terms of Q. In the case where the strategy leads to a derivation that does not terminate on q, the cardinality of $[\zeta](q)$ is 0, the policy is still considered as consistent.

Example 4. Consider the following policy:

$$\begin{array}{rcl} \wp_1 = (\ \mathcal{F}_1 & = & \{g : A \times A {\rightarrow} A, permit : A, deny : A\}, \\ D_1 & = & \{permit, deny\}, \\ R_1 & = & \{g(x, y){\rightarrow}x,\ g(x, y){\rightarrow}y\}, \\ Q_1 & = & g(\mathcal{T}(\mathcal{F}), \mathcal{T}(\mathcal{F})), \\ \zeta_1 & = & \texttt{universal}(R)) \end{array}$$

Then $\wp_1$ is a security policy under the conditions expressed in Definition 1, but it clearly fails to be consistent, since $[\zeta](g(permit, deny)) = \{permit, deny\}$.

Since we assume strategies to be closed by concatenation, confluence under strategy can be simply expressed:

Definition 3 (Confluence under strategy). *A rewrite system R is confluent under a strategy ζ when $\forall u, v_1, v_2 \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\{v_1, v_2\} \subseteq [\zeta](u)$ then $[\zeta](v_1) \cap [\zeta](v_2) \neq \emptyset$.*

If we consider the `universal` strategy, the above definition reduces to the usual one of confluence. Therefore:

Proposition 1. *The policy $(\mathcal{F}, D, R, Q, \texttt{universal})$ is consistent as soon as R is confluent on $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

A second fundamental property is termination:

Definition 4 (Termination). *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is terminating if for every $q \in Q$, all derivations of source q in ζ are finite.*

It is a fundamental property of term-rewriting systems that a system that is terminating confluent enjoys the property than any term evaluates to a unique normal form.

Example 5. The policy from Example 3 is terminating and confluent, which can be easily checked by analyzing the rules in R. This guarantees that the evaluation of any request will return a unique decision.

Example 6. Consider the policy:

$$\begin{array}{rrcl} \wp_2 = (& \mathcal{F}_2 & = & \{a : A, permit : A, deny : A\}, \\ & D_2 & = & \{permit, deny\}, \\ & R_2 & = & \{a \to a, a \to deny\}, \\ & Q_2 & = & \{a\}, \\ & \zeta_2 & = & \texttt{universal}(R)) \end{array}$$

$\wp_2$ is a security policy. In contrast to the previous example, this policy is consistent (since the corresponding rewrite relation is confluent), but it is not terminating.

Some simple sufficient conditions allows us to apply termination results from rewrite theory:

Proposition 2. *A policy $(\mathcal{F}, D, R, Q, \zeta)$ terminates provided that all derivations in ζ are finite or if R is strongly terminating (i.e. all derivations in $\texttt{universal}(R)$ are finite).*

To ensure strong termination, classical quite powerful termination tools can be used like recursive path orderings [13] or dependency pairs [1]. Termination allows one to localize confluence check following Newmann's lemma and this can be made operational via the completion algorithm [26]. Therefore we inherit sufficient condition for policies using the `universal` strategy. Since in general we use the finer notion of termination and confluence under strategies, this opens new research questions to establish sufficient conditions also for rich strategies.

Another expected property of a policy strategy is that it is able to evaluate every incoming request into an authorization term, following its strategy. This is expressed through the completeness property:

Definition 5 (Completeness). *A security policy $\wp = (\mathcal{F}, D, R, Q, \zeta)$ is complete if $\forall q \in Q$, $[\zeta](q) \subseteq D$ and $[\zeta](q) \neq \emptyset$.*

This definition is close to the definition of sufficient completeness of a rewrite system, which states that every ground term evaluates to a term exclusively built with constructors and possibly variables [11, 24]. Several algorithms have been developed to check sufficient completeness or to complete a set of patterns to ensure this property [8].

Proposition 3. *A policy $(\mathcal{F}, D, R, \mathcal{T}(\mathcal{F}), \texttt{universal}(\mathcal{R}))$ is complete provided that D is a set of constructors for R and that R is terminating and sufficiently complete.*

The strong sufficient conditions of this proposition may be relaxed to a weakly terminating system R and an innermost strategy, as shown in [18].

4 Policy Composition

Let us now focus on the problem of *combining* policies in a modular way, relying on the long history of research in combining rewrite systems. This combination consists in taking the union of signatures and rules of the two policies components, choosing the sets of requests and decisions, and building a strategy for the combination of the two strategic rewriting in each component of the composition. However, combining access-control policies naively results in inconsistent or non terminating policies and we show how syntactic conditions and strategies may help to keep these suitable properties for the composition of two policies. Based on the example of XACML policy combiners, we explore the idea of a rich combination language for policies based on rewriting strategies.

4.1 Definition and properties of policy composition

Definition 6 (Policy Composition). *The composition of the two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \zeta_i)$ $(i = 1, 2)$ is the policy $\wp = (\mathcal{F},\ D,\ R,\ Q,\ \zeta)$, where:*

1. $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$;
2. $D_1 \cup D_2 \subseteq D \subseteq \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2)$;
3. $R = R_1 \cup R_2$;
4. $Q_1 \cup Q_2 \subseteq Q \subseteq \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2)$;
5. *ζ is a rewrite strategy for R.*

The main design choices behind this definition are the following: when defining the composition of two policies, we must ensure that the generated policy satisfies the conditions declared in Definition 1; the set of requests for the combined policy contains terms of the form determined by its sub-policies, but may also contain any additional well-formed closed term that can be constructed from the combined policy signature. For example, suppose that $\mathcal{F}_1 = \{0, f\}$, $Q_1 = f(\mathcal{T}(\mathcal{F}_1))$ and $\mathcal{F}_2 = \{g\}$, $Q_2 = g(\mathcal{T}(\mathcal{F}_2))$, then a valid request would be $g(f(0))$; the combination strategy is in charge of defining how the composed policy rewrites request terms. It may or not be built in a modular way by composing ζ_1 and ζ_2. It often can be expressed as a functional composition of component strategies.

Example 7. We take the policy from Example 3, to show how we can extend policies with additional rules. Consider the access control rule R' below:

$$auth(req(phy(x), write, r), urgency)\rightarrow permit$$

A strategy $\zeta' = \texttt{choice}(R', R, auth(q, c)\rightarrow na)$ extends the previous policy by enforcing the rule for urgency cases first, and at the same time does not interfere with the decisions generated by the previous set of rules. This is a direct consequence of the semantics of the `choice` strategy.

The next example illustrates that much care must be taken in composing two policies.

Example 8. Consider the policies $\wp_1$, from Example 4, and the policy $\wp_3$ below.

$$\begin{aligned}
\wp_3 = (\ \mathcal{F}_3 &= \{permit : A, deny : A,\ g : A \times A\rightarrow A,\ f : A \times A \times A\rightarrow A\}\\
D_3 &= \{permit, deny\}\\
R_3 &= \{f(permit, deny, x)\rightarrow f(x, x, x),\\
&\quad f(deny, permit, x)\rightarrow f(x, x, x),\\
&\quad f(x, x, x)\rightarrow x\},\\
Q_3 &= f(\mathcal{T}(\mathcal{F}_2), \mathcal{T}(\mathcal{F}_2), \mathcal{T}(\mathcal{F}_2)),\\
\zeta_3 &= \texttt{universal}(R_2)\ \ )
\end{aligned}$$

The composition $\wp$ of $\wp_1$ and $\wp_2$ can be defined in a straightforward way as $\wp =$:

$$(\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_3, D = D_1 = D_3, R = R_1 \cup R_3, Q = \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_3), \zeta = \texttt{universal}(R))$$

These two policies as clearly terminating and share only symbols $permit$ and $deny$. It is therefore quite intuitive to believe that their composition will be also terminating. But this is false since the following request has an infinite derivation:

$$\begin{aligned}
&f(g(permit, deny), g(permit.deny), g(permit, deny))\rightarrow\\
&f(permit, g(permit, deny), g(permit, deny))\rightarrow\\
&f(permit, deny, g(permit, deny))\rightarrow\\
&f(g(permit, deny), g(permit.deny), g(permit, deny))\ldots
\end{aligned}$$

Many modularity results for confluence and termination of rewrite systems have been produced and the interested reader can refer for instance to [37] for a survey. Confluence and termination are in general not modular properties for rewrite systems. In the context of rewrite system on disjoint signatures, confluence is modular, while termination is not [40]. However, adding syntactic conditions on rewrite rules or existence of a simplification ordering, allows getting positive results. Relying on the results of the rewrite system community [41, 38, 33, 19, 27], we can state the following useful results about composition of security policies.

Proposition 4. *Let us consider two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \texttt{universal})$ ($i = 1, 2$) such that $\mathcal{F}_1$ and $\mathcal{F}_2$ are disjoint and their composition $\wp = (\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \texttt{universal})$. If $\wp_1$ and $\wp_2$ are consistent, then $\wp$ is consistent. If $\wp_1$ and $\wp_2$ are terminating, then so is $\wp$, provided:*

1. *neither R_1 nor R_2 contain collapsing rules, or*
2. *neither R_1 nor R_2 contain duplicating rules, or*
3. *R_1 or R_2 contains neither collapsing rules nor duplicating rules, or*

4. *termination of R_1 and of R_2 are proved by simplification ordering.*

Relaxing the disjointness assumption of signatures in the previous results led to consider constructor-sharing systems [28], composable systems [34] or hierarchical combinations of rewrite systems generalizing the previous ones by allowing a certain sharing of defined symbols [14].

The interest of rewriting strategies appears again in their composition. For instance, in contrast to termination, innermost termination has a nice modular behavior, for disjoint disjoint unions, constructor-sharing systems, composable systems and for certain hierarchical combinations. We can take advantage of such results about innermost termination[20] to state the following result:

Proposition 5. *Let us consider two policies $\wp_i = (\mathcal{F}_i, D_i, R_i, Q_i, \mathtt{innermost})$ ($i = 1, 2$) such that $\mathcal{F}_1$ and $\mathcal{F}_2$ are disjoint or share only constructors, and $\wp$ be their composition $(\mathcal{F}_1 \cup \mathcal{F}_2, D_1 \cup D_2, R_1 \cup R_2, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_2), \mathtt{innermost})$. Then $\wp$ is terminating as soon as $\wp_1$ and $\wp_2$ are.*

Example 9. Let us consider again the policies $\wp_1$, from Example 4 and $\wp_3$, from Example 8, but now with different strategies $\zeta'_1 = \mathtt{innermost}(R_1)$ and $\zeta'_3 = \mathtt{innermost}(R_3)$. Their combination $\wp = (\mathcal{F}_1 \cup \mathcal{F}_3, D, R_1 \cup R_3, \mathcal{T}(\mathcal{F}_1 \cup \mathcal{F}_3), \mathtt{innermost}(R))$ is terminating according to Proposition 5.

4.2 Semantics of XACML Policy Combiners

In this section, we give an executable semantics of the composition operators of XACML [36] using the formalism proposed above. Using rewriting and strategies it is possible to give such a semantics to Core-XACML [16], by translating a its rule sets into rewrite rules. We do not detail this process in this paper due to the lack of space. Basically, these operators were designed to disambiguate different decisions that a policy (or a set of policies) may generate. The combiners described in the XACML specification are[5]:

– *permit-overrides*: whenever *one* of the policies answers to a request with a *granting* decision, the final authorization for the composed policy will be granted. The policy will generate a *denial* only in the case at least one of the sub-policies denies the request, and all others return *not-applicable*.
– *deny-overrides*: this combiner has a similar semantics to *permit-overrides*, with the difference that denials takes precedence.
– *first-applicable*: the decision produced by the combined policy corresponds to the authorization determined by the first sub-policy that does not fail, and whose decision is different from *not-applicable*.

Consider the signature and query terms of Example 3, and the set of rules below:

$$\begin{array}{lcll}
p_1 & : & auth(req(p, write, record(x)), respPhy(p, patient(x))) & \to permit \\
p_2 & : & auth(req(phy(p), write, r), c) & \to deny \\
p_3 & : & auth(req(phy(p), write, r), urgency) & \to permit \\
p_4 & : & auth(q, c) & \to na
\end{array}$$

[5] Additionally, the specification brings the “only-one-applicable” and ordered versions of these policy combiners.

These rules are clearly overlapping: since the variable c captures different conditions for access to the medical record. A naive way of translating the conflict resolution operators of XACML into rewriting strategies, is to take into account the order of the rules presented above, and use the strategy `choice`:

$$\begin{aligned}\zeta_{po} &= \texttt{choice}(p_1, p_3, p_2, p_4)\\ \zeta_{do} &= \texttt{choice}(p_2, p_1, p_3, p_4)\\ \zeta_{fa} &= \texttt{choice}(p_1, p_2, p_3, p_4)\end{aligned}$$

However, this encoding of XACML operators would work only for this example, and would not have the expected results for other policies like for instance in Example 8. This is due to the fact that the right-hand sides of the rules are variables, and we cannot decide which priority order for the rules on a strategy will overrid e with permit, or deny decision.

A general encoding of these conflict resolution operators require more advanced strategies. A raw encoding of the *permit-overrides* operator as a rewriting-based policy simply takes the strategy components as arguments, as defined below (*deny-overrides* can be encoded in a similar way):

$$[\zeta_{po}(\zeta_1, \zeta_2)](q) = \begin{cases} \{permit\} & \text{if } ([\zeta_1](q) \ = \ \{permit\} \vee [\zeta_2](q) \ = \ \{permit\}) \\ \{deny\} & \text{if } ([\zeta_1](q) \ = \ \{deny\} \vee [\zeta_2](q) \ = \ \{deny\}) \\ & \quad \wedge ([\zeta_1](q) \ = \ \{na\} \vee [\zeta_2](q) \ = \ \{na\}) \\ \{na\} & \text{if } ([\zeta_1](q) \ = \ \{na\} \wedge [\zeta_2](q) \ = \ \{na\}) \end{cases}$$

A more natural encoding using strategic rewriting is simply:

$$\begin{aligned}[\zeta_{po}(\zeta_1, \zeta_2)](q) = \texttt{choice}(\ & \texttt{seq}(\zeta_1(q), permit{\to}permit),\\ & \texttt{seq}(\zeta_2(q), permit{\to}permit),\\ & \texttt{seq}(\zeta_1(q), deny{\to}deny),\\ & \texttt{seq}(\zeta_2(q), deny{\to}deny),\\ & \zeta_1(q), \zeta_2(q))\end{aligned}$$

It works as follows: the `choice` strategy ensures that only its first succeeding argument will be returned. Its first argument $\texttt{seq}(\zeta_1(q), permit{\to}permit)$ evaluates in sequence $\zeta_1(q)$. In case it evaluates to *permit*, the rule $permit{\to}permit$ applies and trivially returns *permit*, otherwise the application of $permit{\to}permit$ fails and therefore the full strategy fails.

The strategy *first-applicable* can be encoded in a similar fashion: We first check if applying the strategies result in $permit$ or $deny$, and we choose the first strategy that returns one of these results. The *not-applicable* decision will always cause the strategy `seq` to fail, unless it is the only decision generated by the sub-policies.

$$\begin{aligned}[\zeta_{fa}(\zeta_1, \zeta_2)](q) = \texttt{choice}(\ & \texttt{seq}(\zeta_1(q), permit{\to}permit),\\ & \texttt{seq}(\zeta_1(q), deny{\to}deny),\\ & \texttt{seq}(\zeta_2(q), permit{\to}permit),\\ & \texttt{seq}(\zeta_2(q), deny{\to}deny),\\ & \zeta_1(q), \zeta_2(q))\end{aligned}$$

In the examples above, XACML operators for combining policies can be encoded in this unambiguous manner.

5 Related and further works

Several approaches define logic languages for access control [17, 21, 22, 23], each focusing on different aspects of access control, e.g. roles or obligations, and providing different levels of expressivity. Some of them propose policy composition operators to disambiguate conflicting policy decisions, like in XACML. In these works, access control is basically defined as inferring the truth value of a certain predicate of the form $\texttt{access}(\texttt{subject}, \texttt{action}, \texttt{object})$ in the underlying logic. We claim that rewriting provides a suitable theoretical and practical framework for expressing flexible access control rules. The strength of the approach relies on the expressivity of strategic rewriting, on the logical background of rewriting logic [31], on the computational efficiency of rewrite rules, and on the existence of several theoretical results and tools, which are readily applicable in a consistent way to access control policies.

With respect to policy composition, a number of works have a close relationship with the formalism introduced here. Bonatti et al. [7] address the composition problem through an algebra of composition operators that is able, for example, to define policy templates, among other operations. The operator definitions can be adapted to several languages and situations since their definition is orthogonal to the underlying authorization language. Basically, in [7], a policy is a set of ground decisions, or authorization terms, and the composition operators are defined over the sets of decisions, but not on the rules used to derive them, which is the originality of our work. Other works follow the same direction, with slight differences on the modeling of policies [43]. Another existing alternative for composing access control policies is implemented by the Polymer system [6], which proposes rather classical operators on policies (conjunction, precedence, etc), and that allows reusing the policy objects, modifying them by executing additional actions, in order to specialize or enforce the policy. Finally, in [30], an interesting approach for policy composition is taken, based on the non-monotonic properties of defeasible logic (where defeasible rules are used to draw conclusions that can be later invalidated). Authors show how this can be used to encode meta-policies describing the content of the security requirements and how the policy is to be combined in the case of composition. This way, a single operator is proposed, which takes into account a precedence relation among the policies. We advocate that this kind of composition can also be achieved using rewriting strategies, by defining priorities on the rules, in the same way as in defeasible logic.

In comparison, our main contributions in this paper are first, to provide a formal definition for access control policies using term rewriting that allows us to describe flexible policies and that supports reasoning about properties like consistency, completeness and termination; and second to give the formal semantics for composition operators in an uniform manner using rewrite strategies.

A first approach for the implementation of rewrite-based policies is to design them in a rewrite environment such as MAUDE or ELAN, in order to get a prototype and study their behavior and properties. The next step is to embed policies in a target language, such as JAVA, using the formal island approach [4]. The Tom system supports the compilation process of our access control policies into JAVAclasses, that in turn can be instantiated in any application to evaluate access requests to sensitive resources, given that a mapping (called “anchoring” in the Formal Island metaphor), linking these objects to the symbols of signature on the policy level is provided.

References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[2] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom Manual. LORIA, Nancy (France), version 2.4 edition, October 2006.

[4] E. Balland, C. Kirchner, and P.-E. Moreau. Formal islands. In M. Johnson and V. Vene, editors, *AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2006.

[5] S. Barker and M. Fernández. Term rewriting for access control. In E. Damiani and P. Liu, editors, *DBSec*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 305–314. ACM, 2005.

[7] P. A. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002.

[8] A. Bouhoula. Spike: a system for sufficient completeness and parameterized inductive proofs. In A. Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 836–840. Springer, 1994.

[9] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:427–498, May 2001.

[10] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[11] H. Comon. Sufficient completness, term rewriting systems and ”anti-unification”. In J. H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1986.

[12] A. S. de Oliveira. Rewriting-based access control policies. In M. Fernandez and C. Kirchner, editors, *Proceedings of the 1st International Workshop on Security and Rewriting Techniques - SecRet'06*, June 2006.

[13] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987.

[14] N. Dershowitz. Hierarchical termination. In *Proceedings 4th International Workshop on Conditional Term Rewriting Systems, Jerusalem (Israel)*, volume 968 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 1994.

[15] S. D. C. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In S. Bhalla, editor, *DNIS*, volume 3433 of *Lecture Notes in Computer Science*, pages 225–237. Springer, 2005.

[16] D. J. Dougherty. Core XACML and term-rewriting systems. Unpublised, March 2007.

[17] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[18] I. Gnaedig and H. Kirchner. Computing constructor forms with non terminating rewrite programs. In A. Bossi and M. J. Maher, editors, *PPDP*, pages 121–132. ACM, 2006.

[19] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd Algebraic and Logic Programming Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, September 1992.

[20] B. Gramlich. On proving termination by innermost termination. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 93–107. Springer-Verlag, July 1996.

[21] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *CSFW*, pages 187–201. IEEE Computer Society, 2003.

[22] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[23] A. Kalam, R. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization based access control. *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131, 2003.

[24] D. Kapur, P. Narendran, D. J. Rosenkrantz, and H. Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Inf.*, 28(4):311–350, 1991.

[25] C. Kirchner, H. Kirchner, and M. Vittek. Designing clp using computational systems. In P. V. Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 8, pages 133–160. MIT press, 1995.

[26] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[27] M. Kurihara and A. Ohuchi. Modularity of simple termination of term rewriting systems. *Journal of IPS Japan*, 31(5):633–642, 1990.

[28] M. Kurihara and A. Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theor. Comput. Sci.*, 103(2):273–282, 1992.

[29] B. Lampson. Protection. ACM Operating Systems Review. *Vol*, 8:18–24, 1974.

[30] A. J. Lee, J. P. Boyer, L. Olson, and C. A. Gunter. Defeasible security policy composition for web services. In M. Winslett, A. D. Gordon, and D. Sands, editors, *FMSE*, pages 45–54. ACM, 2006.

[31] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[32] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for maude. *Electr. Notes Theor. Comput. Sci.*, 117:417–441, 2005.

[33] A. Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 396–401, 1989.

[34] A. Middeldorp and Y. Toyama. Completeness of combinations of constructor systems. In *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, 1991. also Report CS-R9058, CWI, 1990.

[35] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2003.

[36] T. Moses. eXtensible Access Control Markup Language (XACML) version 2.0. Technical report, OASIS, 2005.

[37] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[38] M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.

[39] Terese. *Term Rewriting Systems*. Cambridge University Press, 2002.

[40] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. Technical report, NTT Electrical Communications Laboratories Japan, 1987.

[41] Y. Toyama. On the church-rosser property for the direct sum of term rewritig systens. *Journal of the ACM*, 34(1):128–143, January 1987.

[42] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[43] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, 2003.