Rewriting-Based Access Control Policies

Anderson Santana de Oliveira

Rewriting-Based Access Control Policies

Anderson Santana de Oliveira [1,2]

INRIA & LORIA
615, Rue du Jardin Botanique, 54600 Villers-lès-Nancy, France

Abstract

In this paper we propose a formalization of access control policies based on term rewriting. The state of the system to which policies are enforced is represented as an algebraic term, what allows to model many aspects of the policy environment. Policies are represented as sets of rewrite rules, whose evaluation produces deterministic authorization decisions. We discuss the relation between properties of term rewriting systems and those important for access control, and the impact of composing policies to these properties.

Key words: Access Control Policies, Term Rewriting Systems

1 Introduction

Term rewriting [1] is a well-established paradigm for specifying, and prototyping systems. It has been proved useful as the theoretical foundations in theorem proving, program transformation, and algebraic specification. In many practical situations, its straightforward formal background allowed to rapidly prototype and verify diverse kinds of systems. In the domain of computer security, term rewriting has been successfully applied to help reasoning about many of its aspects, notably in verification of security protocols [6,18].

Nevertheless, there are not many applications of term rewriting to access control policies: up to our knowledge, few approaches have tried to introduce the use of rewriting into this domain. The reader is referred to see a more extensive discussion on related work in section 7.

Access control concerns stating which *actions*, that *principals (or subjects)* are allowed to execute in order to manipulate the *objects (or resources)* of a given system. The most widespread solution to this problem is to use an *access control matrix*, a model adopted in the design of several operating systems. The lines of the matrix enroll the subjects, the columns list the resources

[1] santana@loria.fr
[2] Supported by Capes BEX.2120.03-8.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of the system, and cells contain what rights (*read, write, execute, ...*) are assigned to each case. A request from a subject to perform a certain action over an object will be granted only if there exists an entry for that action in the column and line regarding those subject and object.

Many models for access control rely on some modified version of the access control matrix. For example, military security policies [4] add a confidentiality levels to subjects and objects, them the (fixed) security policy states that a subject which cannot write to an object with inferior security level, and that it cannot read from objects with superior security levels - *no reads-up, no writes-down*. Even though the access control matrix is a solution to many access control models, it is not appropriate to capture more dynamic policies, such as policies that depend on time, location, and many other possible attributes of the policy environment.

This paper presents a formalization of access control policies based on term rewriting. Policies are represented as sets of rewrite rules, whose evaluation produces authorization decisions, whilst requests and the environment where policies are enforced are represented as algebraic terms. Since we consider the policy environment as a "database of facts" under the form of a term, this formalization allows us to capture many dynamic aspects that are important for policy enforcement, e.g. diverse attributes of subjects and resources, referred as content-dependent conditions in the literature [8].

The main goals are to provide a formal semantics for an expressive access control policy language, and to use standard methods for checking properties of the term rewriting systems associated to the policies, which would increase the trust in the security policy. An example of such properties are absence of conflicts - no grant and deny are assigned to the same access request. One advantage of associating properties of term rewriting systems to desired properties of a policy is that one can check whether the property holds for policy composition. There are well-known positive results on the preservation of termination under union of term rewriting systems, for example.

Another goal of this formalization is to be able to facilitate policy enforcement. The architecture we propose here clearly separates policy and enforcement mechanism. Since policies are rewrite rules, a standard rewrite engine can do the job of applying the policy to requests and evaluating the results.

The paper is organized as follows: Section 2 recalls some useful definitions on term rewriting systems, Section 3 illustrates by an example what kind of access control policy we want to express, Section 4 presents what are the elements of the policy environment, Section 5 presents and discuss rewriting-based policies, Section 6 describes what kind of security mechanism is necessary to enforce these policies, Section 7 presents a discussion on related works, and Section 8 concludes and points out some future developments.

2 Preliminaries on Term Rewriting Systems

We recall some basic definitions on signatures and terms. A signature $\Sigma = \{\mathcal{S}, \mathcal{F}\}$ is a set of sorts $\mathcal{S}$, together with a set of function symbols, each one associated to a natural number by the arity function (ar : $\mathcal{F} \to \mathbb{N}$). $\mathcal{F}_n$ is the subset of function symbols having n for arity, $\mathcal{F}_n = \{f \in \mathcal{F} \mid \text{ar}(f) = n\}$. $\mathcal{T}(\Sigma, \mathcal{X})$ is the set of *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 $\mathcal{T}(\Sigma)$ is the set of ground terms.A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\Sigma)$, written, when its domain is finite, $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$.

A rewrite rule is an ordered pair of terms denoted $l \to r$, where l, $r \in \mathcal{T}(\Sigma, \mathcal{X})$, and $l \notin \mathcal{X}$. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A rewrite system or term rewriting system is a (finite or infinite) set of rewrite rules.

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

A subterm $t_{|\omega}$ where the rewriting step is applied is called *redex*. A term that has no redex is said to be irreducible for R or in *R-normal form*.

A rewrite derivation is any sequence of rewriting steps $t_1 \to_R t_2 \to_R \ldots$ A rewrite derivability relation $\overset{*}{\longrightarrow}_R$ is defined on terms: $t \overset{*}{\longrightarrow}_R t'$ if there exists a rewriting derivation from t to t'. If the derivation contains at least one step, it is denoted by $\overset{+}{\longrightarrow}_R$. A term rewriting systems is terminating if all reduction sequences are finite. It is confluent if for all terms t, u ,v, $t \overset{*}{\longrightarrow}_R u$ and $t \overset{*}{\longrightarrow}_R s$ implies $u \overset{*}{\longrightarrow}_R s$ and $v \overset{*}{\longrightarrow}_R s$, for some s.

Some researchers have been interested in analyzing whether the properties mentioned in the previous paragraph are modular. Let R_1 and R_2 be two rewrite systems. A property $Prop$ is modular if when $Prop$ is satisfied by R_1 and R_2, then $Prop$ is satisfied by the union of R_1 and R_2. Most of the positive results on the union of term rewriting systems assume the signature of the composed systems to be disjoint [3]. We denote the union of term rewriting systems with disjoint signatures by the operator $\oplus$. Confluence is a modular property of rewrite systems with disjoint signatures [21]. Termination, however, is not [20]. In order to obtain positive results for termination, it is necessary to impose additional syntactic conditions on the rewrite rules. A rewrite rule $l \to r$ is a collapsing rule if r is a variable. A rewrite rule $l \to r$ is said duplicating if there exists a variable that has more occurrences in r than in l. Given these definitions, we can recall the following results:

Let R_1 and R_2 be two terminating rewrite systems, then

(i) If neither R_1 nor R_2 contain collapsing rules, then $R_1 \oplus R_2$ is terminating.

[3] The disjointness assumption can be relaxed in the case of constructor systems, if only constructors are shared by the component signatures [14].

(ii) If neither R_1 nor R_2 contain duplicating rules, then $R_1 \oplus R_2$ is terminating.

(iii) If one of the systems R_1, R_2 contains neither collapsing rules nor duplicating rules, then $R_1 \oplus R_2$ is terminating.

(1) and (2) are proved in [17]. (3) is proved in [13].

A survey on modularity of various properties of term rewriting systems is found on [11].

3 Motivating Example

The example that follows has been slightly adapted from the XACML specification [15], an initiative from the Oasis group to create a standard markup language for access control. Let us suppose that a medical corporation adopts the policy below:

(i) A person, identified by his or her patient number, may read any record for which he or she is the designated patient.

(ii) A person may read any record for which he or she is the designated parent or guardian, and for which the patient is under 16 years of age.

(iii) A physician may write to any medical element for which he or she is the designated primary care physician.

(iv) An administrator shall not be permitted to read or write to medical elements of a patient record.

An application running in the computers of such corporation must ensure that this policy is respected. Therefore, access control must be applied in every scenario of the application execution. This is costly, and additionally, it is not related to the main functionalities which is to electronically manage the medical business in question.

A possible execution of this system is shown in Figure 1. At a given point of time, the system is in state s_i, whose associated database of facts appears in the first line of the table. Then, the primary physician of a certain patient record, requires to prescribe a new medical element for this patient (see the second line of the table). Since this request is allowed, according to the policy we just described, the system state changes to state s_{i+1} (last line of the table), containing the updated entry for the medical record.

Here, we assume that the application delivers its state, under the form of a term, as well as the user requests, to a reference monitor. In practice, this will require to modify the program in order to capture its state, and to intercept the control flow for monitoring intervention, every time a resource is to be accessed.

s_i	patient("Bart Simpson", 1, 14, guardian("Homer Simpson")) + record(patient("Bart Simpson", 1, 14, guardian("Homer Simpson")), physician("Julius Hibbert", 1), antibiotic, payment(visa)) + physician("Julius Hibbert", 1)
Request	request (physician("Julius Hibbert", 1), writeMedicalElements , record(patient("Bart Simpson", 1, 14, guardian("Homer Simpson")), physician("Julius Hibbert", 1) , antibiotic, payment(Visa)))
s_{i+1}	patient("Bart Simpson", 1, 14, guardian("Homer Simpson")) + record(patient("Bart Simpson", 1, 14, guardian("Homer Simpson")), physician("Julius Hibbert", 1), antibiotic and aspirin, payment(visa)) + physician("Julius Hibbert", 1) .

Figure 1. A system state transition after a request to write a medical element

4 The Policy Environment

Regarding access control, one is interested in stating which *actions*, *principals (or subjects)* are allowed to execute in order to manipulate the *objects (or resources)* of a given system. The most widespread solution to this problem is to use an *access control matrix* where lines list the subjects, columns contain the objects and cells keep information about the rights (*read, right, execute*) assigned to each case.

This schema is enough to address most of the requirements for mandatory and discretionary access control models, but it is not adequate to express higher level policies, like the one from our running example, presented in section 3. For declaring this kind of policy, it is necessary to write sentences about the current values of attributes of subjects and resources, and not only their identities. We call *policy environment* the configuration of all elements relevant to access control.

An arbitrary application, that must respect a given security policy, called *target system* and denoted T, is represented by a set of states and state transitions, which are triggered by access requests. To each state s_i of T we associate an algebraic term containing the facts that are true in s_i. Requests are also represented as terms (see Figure 1).

The idea we defend here, which was recently exposed in some papers [3,16], is that access control is one aspect of the application, that can be specified, implemented and maintained independently. Furthermore, the mechanism applying a certain policy should be an external entity with respect to the application. An overview picture of policy enforcement is presented in Figure 2. The application context in state s_i together with the current request $R(S, A, O)$ are delivered to a reference monitor. The reference monitor then evaluates the request according to the policy. In the case the policy grants

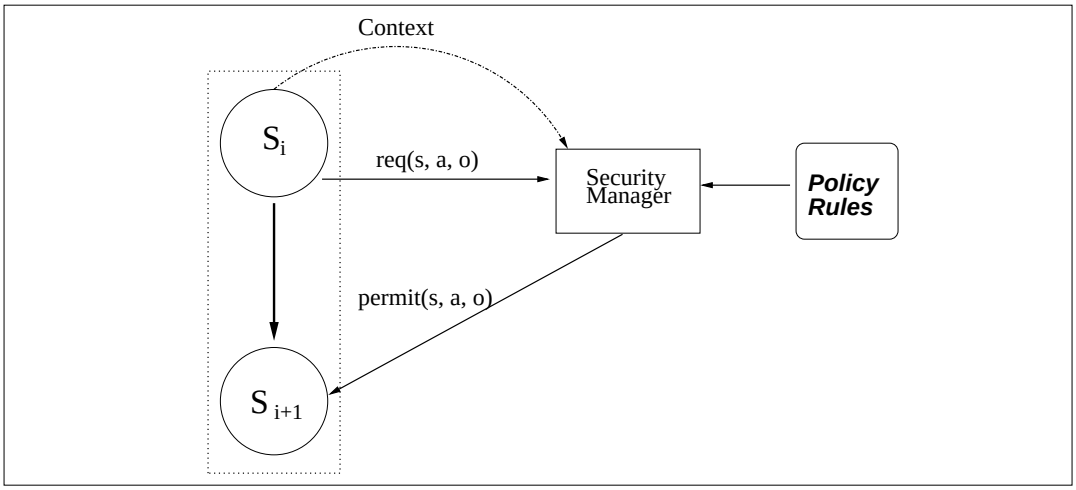

Figure 2. General representation schema

access for the request, the application proceeds.

We note Σ_T the signature of the target system T, which provides the profiles of the constructors to build the representation of the system state. Consequently, the database of facts at each stage of the execution of T is a the set ground terms from $\mathcal{T}(\Sigma_T)$.

In the example that follows, we show the signature of the terms that appear in Figure 1. We have used order-sorted specifications to formalize the problem and the system using Maude [7].

Example 4.1 We use the signature below for medical system of section 3, where a patient is represented by a term recording his name, number, age, and a guardian, in this order.

```
fmod MEDICAL-SYSTEM-SIGNATURE is
      protecting STRING .
      protecting NAT .
         sort  Patient Physician Record Administrator Guardian
               MedicalElements OtherElements .
         op patient  : String Nat Nat Guardian -> Patient [ctor] .
         op administrator : Nat -> Administrator [ctor] .
         op guardian : String -> Guardian [ctor] .
         op physician : String Nat -> Physician [ctor] .
         op record : Patient Physician MedicalElements
              OtherElements -> Record [ctor] .
endfm
```

In order to better express policies, it is necessary to indicate which sorts from Σ_T are subsorts of subject, action and object, as well as to introduce the available actions in T.

Example 4.2 The module below illustrates how we can determine subjects, objects and actions in the medical system presented in Section 3, through the use of subsorts. Actions are represented as constant symbols for simplicity.

```
fmod MEDICAL-SYSTEM-TERM-SIGNATURE is
            including MEDICAL-SYSTEM-SIGNATURE .
            including POLICY-SIGNATURE .
            subsort Physician < Subject .
            subsort Patient < Subject .
            subsort Guardian < Subject .
            subsort Administrator < Subject .
            subsort Record < Object .
            subsort MedicalElements < Object .
            subsort OtherElements < Object .
            op readRecord : -> Action .
            op writeRecord : -> Action .
            op readMedicalElements : -> Action .
            op writeMedicalElements : -> Action .
            op readOtherElements : -> Action .
            op writeOtherElements : -> Action .
endfm
```

This representation allows us to distinguish subjects and objects among the subterms appearing in the conjunction of ground terms of the target's application current state.

5 Rewriting-Based Policies

In this section, we address the problem of specifying access control policies through term rewriting systems. We present an initial definition of security policy that characterizes policies syntactically. This definition includes potentially "unsafe" policies. After discussing the desired properties of an access control policies, we present a refined definition for "safe" policies.

A security policy is a statement of what is, and what is not, allowed [5]. When dealing with access control, a (formal) policy specification language will help to unambiguously define rules about what actions principals are allowed to execute over a set of resources. Consequently, a policy specification language must encode high level statements into a function from access requests to authorization decisions.

Let the signature for rewriting-based access control policies be stated as follows. Requests are represented as ground terms containing the 3-tuple subject, action and subject, using the following constructor symbol

$$req : Subject \times Action \times Object \rightarrow Request$$

Authorized situations are separated from unauthorized ones through the use of distinct constructors for each case. The signatures for authorization terms are

$$\{deny, permit\} : Subject \times Action \times Object \rightarrow Authorization$$

The goal of the policy designer is to provide a set of rules that will be used by the reference monitor to evaluate every incoming request. The resulting

decision does not depend exclusively on the request, but also on the context of the target application at the time the request is made. The rules of a policy will define the *auth* operator, whose signature is

$$auth : Request \times Term \rightarrow Authorization$$

The *auth* function returns a decision (*permit* or *deny*) about a given request, based on the information contained in the request and in the database of facts, the $Term$ argument, which is a conjunction of ground terms using an associative and commutative operator. The full policy signature, Σ_P, is illustrated by the code that follows.

```
fmod POLICY-SIGNATURE is
        sort Object .
        sort Subject .
        sort Action .
        sort Term .
        sort Request .
        sort Authorization .
        subsort Subject < Term .
        subsort Object < Term .
        subsort Action < Term .
        op req : Subject Action Object -> Request [ctor] .
        op permit : Subject Action Object -> Authorization [ctor] .
        op deny : Subject Action Object -> Authorization [ctor] .
        op auth : Request Term -> Authorization .
        op _+_ : Term Term -> Term [assoc comm ] .
endfm
```

Definition 5.1 [Security Policy] An access control security policy, $\mathcal{P}$, is a term rewriting system over $\mathcal{T}(\Sigma, \mathcal{X})$, with $\Sigma = \Sigma_T \cup \Sigma_P$, where the top symbol of the left hand side of each rule is the *auth* function.

This definition imposes syntactical restrictions on the form of the rewrite rules, in such way that forces the set of rules of a policy to define the *auth* function. The example that follows illustrates how a policy can be declared.

Example 5.2 The following set of rewrite rules translates the natural language rules from Section 3 in our formalism.

```
mod POLICY1 is
    protecting MEDICAL-SYSTEM-TERM-SIGNATURE .
    var p : Patient .
    var ph : Physician .
    var g : Guardian .
    var adm : Administrator .
    var me : MedicalElements .
    var oe : OtherElements .
    vars s1 s2 : String .
    var t : Term .
    var n1 n2 : Nat .
```

```
    rl [patReadRecord] : auth( req(p, readRecord,
      record(p, ph, me, oe)), p + record(p, ph, me, oe)
      + t ) => permit(p, readRecord, record(p, ph, me, oe)) .

    rl [guardReadRecord] : auth( req( g, readRecord,
      record(patient(s1, n1, n2, g), ph, me, oe )),
      patient(s1, n1, n2, g) + record(patient(s1, n1, n2, g),
      ph, me, oe) + t) => permit(g, readRecord,
      record(patient(s1, n1, n2, g), ph, me, oe)) .

    rl [physWriteMedElem] : auth( req( ph, writeMedicalElements,
      record(p, ph, me, oe)), record(p,ph, me, oe) + t)
      => permit(ph, writeMedicalElements, record(p, ph, me, oe)) .

    rl [admReadMedElem] : auth(req( adm, readMedicalElements,
      record(p, ph, me, oe)), adm + record(p,ph, me, oe) + t)
      => deny(adm, readMedicalElements, record(p, ph, me, oe)) .

    rl [admWriteMedElem] : auth( req( adm,writeMedicalElements,
      record(p, ph, me, oe)), adm + record(p,ph, me, oe) + t )
      => deny(adm, writeMedicalElements, record(p, ph, me, oe)) .
endm
```

This is a straightforward implementation of the policy stated in the running example of this paper. It covers all cases mentioned in the informal rules presented previously. There are many issues that can complicate policy enforcement. For example, the rule application may not terminate, causing the target system to be blocked waiting for an authorization. In the next section we discuss what are the desired properties for rewriting-based policies.

5.1 Properties of security policies

Termination

The first interesting property is termination. This will ensure that every request evaluation is finite, thus avoiding the target application execution to block indefinitely. Termination of term rewriting systems has been widely studied and there are many tools available that check termination of term rewriting systems such as CiMe, Approve, Cariboo, to mention a few [4]. The idea is that a refinement discipline for policy deployment has to be followed by users, and that many verification steps have to be applied as needed, before enforcing some policy.

Absence of conflict

The combined use of positive and negative authorizations brings two main problems: incompleteness, when no authorization is specified for a certain

[4] Check for references and results on the Termination Competition home page: http://www.lri.fr/~marche/termination-competition/

request, and inconsistency, when for an access there are both negative and positive authorizations. Classical approaches for policy specification adopt either *closed policy* or *open policy* assumption, meaning that only positive or negative authorizations need to be specified, respectively. This has shown to be restrictive in practice. The current trend is to allow the user to discriminate between what is and what is not allowed [8].

In the case of rewriting-based policies, conflicts can be avoided if the corresponding rewriting system has the confluence property. This will ensure that a single response is derived from a given request and from the application current state.

Some policy specification languages propose *conflict resolution strategies*, that assign priorities to the conflicting cases. For example, one can say that *deny overrides* any other parallel authorization computed for a certain request. Another example, that sounds more difficult to implement is *most specific overrides* that requires to provide an order among actions and objects.

Completeness

Completeness is usually achieved by assuming that one of either the open or closed policy operates as a default. By using an incremental discipline, the user can early detect this problem by using tools that are directed to checking the *sufficient completeness* of term rewriting systems. A term rewriting system is called complete if all ground terms can be reduced to a normal form that only contains constructors. Furthermore, in the case of our policies, the more basic algorithms can be applied, since the form of the rules is simpler than in the general case: we do not have conditions, nor constraints and the possible normal forms are known in advance, they must be either *permit* or *deny* terms.

Alternatively, the user can make use of rules determining the default case to be applied in the case no redex exists for a request. These rules are either of the form

$$auth(req(s_1, a_1, o_1), t) \rightarrow deny(s_1, a_1, o_1)$$
or
$$auth(req(s_1, a_1, o_1), t) \rightarrow permit(s_1, a_1, o_1)$$

for closed or open policy respectively. These rules must be enforced with the help of a strategy that says this rule will be applied only when all other ones fail.

Given this discussion on the desired properties of an access control security policies, we are ready to introduce the following definition:

Definition 5.3 [Trusted Security Policy] A trusted access control security policy is a *terminating* and *confluent* term rewriting system, $\mathcal{P}$, on the signature $\Sigma = \Sigma_T \cup \Sigma_P$, that *completely* defines the *auth* function.

The word trust was chosen to express that the system administrator can have much more confidence in a rewriting-based policy which has the proper-

ties of termination , confluence and sufficient completeness. This means that the policy unambiguously states authorizations.

5.2 Policy composition

Policy composition is an issue with increasingly interest nowadays. The main problems associated with composition are related with the following question: what can we expect when two companies, departments, etc, decide to put their policies together to work on certain projects. Assumed that policies have been specified and developed correctly in isolation, and they are free of ambiguities and conflicts, the question is to know whether the new joint policy will retain these desired characteristics.

Some approaches, like XACML [15] and Polymer [3] adopt composition operators that are responsible for solving conflicts. The operator has a built-in strategy for choosing what decision must be taken in case of conflicting request responses. In this preliminary work, we do not propose composition operators, but we consider some verification that can be performed to preserve trusted policies, as stated in Definition 5.3.

Since trusted policies are terminating and confluent term rewriting systems, which ensures the good properties of being absent of conflicts and deterministic, one can check the syntactical restrictions exposed in Section 2.

6 Security Mechanisms

In this section we present how security mechanisms can enforce rewriting-based policies, and how systems can be considered secure with respect to this formalization.

A *Security mechanism* ensures that a target system do respect the policy being enforced during its whole execution. A state transition of T, $s_i \mapsto s_{i+1}$, corresponds to an access request from a subject to execute an action over a resource. The security mechanism must apply the rewrite rules provided by a policy $\mathcal{P}$, over the terms of T and the current request, $auth(req(s, a, o), t)$. In the case it evaluates to $permit(s, a, o)$, the computation of T can continue, if it evaluates to $deny(s, a, o)$ then the enforcement mechanism must abort the execution of T, this characterizes an *execution monitoring* security mechanism [19].

A given state t_i of a target's execution is considered valid if the information contained in that state is authentic, which means that the database of facts is not modified by an external malicious entity, and that this state was reached through a sequence of positive authorizations.

Definition 6.1 [Secure System] A target system T is said secure w.r.t. a policy $\mathcal{P}$, if it starts from a valid state t_0, and for every transition state $t_i \mapsto t_{i+1}$ a new valid state is produced.

This definition is close to classical automata-based approaches of secure systems, from Goguen and Meseguer [10], and more recently, Scheneider [19], where assertions are stated about the possible execution paths of the target system.

7 Related Work

The works more closely related with the one described in this paper are recent initiatives that introduce term rewriting to the specification of security policies. In [9], term rewriting is used to control the confidentiality level of data, by describing downgrading functions. Whilst in [2], authors model access control lists and role based access control as term rewrite systems. They characterize consistency, totality, and completeness of policies w.r.t the properties of the rewriting systems defining them. Therefore, the second approach shares some of the goals of this paper, with the difference that we focus on the dynamic aspects of the policy environment to specify authorizations, while their work is directed to capturing the access control lists and role-based access control models.

Most of the formal approaches based on rules to specify security policies concern some dialect of a logic language. The reasons are clear: logic languages have formal semantics, they are suitable for implementation and validation, thus facilitating policy verification. One of the most representative work on this line of research is presented in [12] where a proposal for a logic-based language that attempts to join expressiveness and performance is made. It allows representing different policies and protection requirements, since it includes four main ingredients: explicit authorizations, using an access control matrix (called authorization table); policy propagation rules, to control access that is indirectly derived from the authorization table; conflict resolution operators; and a policy language for specifying decisions. Authorization specifications are stated as logic rules defined over a small set of predicates. In order to keep complexity under control the format of the rules is restricted. The authors of [12] present a materialization technique for producing, storing, and updating the stable model of the policy. The model is computed on the initial specifications and updated incrementally.

With respect to policy enforcement, the work presented here has relationships with the Polymer system [3]. Polymer adopts execution monitoring to enforce access control on Java programs. Policies are first-class objects structured to be arbitrarily composed with other policies. Program monitors, are formalized as abstract machines, called edit automata, that examine the sequence of application program actions and transform the sequence when it deviates from a specified policy. A policy compiler compiles program monitors defined in the Polymer language into plain Java and then into Java bytecode. Then, a second tool is a bytecode rewriter that processes ordinary Java bytecode, inserting calls to the monitor in all the necessary places.

8 Conclusions and Future Work

We have discussed in this paper a formalization for access control policies using term rewriting. The policy language allows users to declare rules which once matched against the target system's current state produce an authorization decision. The language is expressive enough to capture different conditions important for defining authorizations in access control. This work sets a basis for applying the well-known techniques developed by the term rewriting community to reason about access control, since we have established a correspondence between the properties of policies and those of term rewriting systems.

As future work, we shall investigate how to solve conflicts using rewriting strategies, and how we can deal with composition of access control policies under the term rewriting perspective. Additionally, we shall build prototypes that validate this model.

9 Acknowledgments

I would like to thank my advisors Claude and Hélène Kirchner for the fruitful discussions on this subject, and also Judson Santiago and Horatiu Cirstea for reading previous versions of this paper.

References

[1] Baader, F. and T. Nipkow, "Term Rewriting and All That," Cambridge University Press, 1998.

[2] Barker, S. and M. Fernandez, *Term rewriting for access control*, in: *Proceedings of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec'2006)*, LNCS, 2006.

[3] Bauer, L., J. Ligatti and D. Walker, *Composing security policies with polymer*, in: *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 305–314.

[4] Bell, E. D. and L. J. LaPadula, *.secure computer systems: Mathematical foundations*, Technical Report Mitre Report ESD-TR-73-278 (Vol. I-III), Mitre Corporation (1974).

[5] Bishop, M., "Introduction to Computer Security," Addison Wesley, 2004.

[6] Cirstea, H., *Specifying authentication protocols using rewriting and strategies.*, in: I. V. Ramakrishnan, editor, *PADL*, Lecture Notes in Computer Science **1990** (2001), pp. 138–152.

[7] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: specification and programming in rewriting logic.*, Theor. Comput. Sci. **285** (2002), pp. 187–243.

[8] di Vimercati, S. D. C., P. Samarati and S. Jajodia, *Policies, models, and languages for access control.*, in: S. Bhalla, editor, *DNIS*, Lecture Notes in Computer Science **3433** (2005), pp. 225–237.

[9] Echahed, R. and F. Prost, *Security policy in a declarative style*, in: *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2005), pp. 153–163.

[10] Goguen, J. A. and J. Meseguer, *Security policies and security models.*, in: *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[11] Gramlich, B., *On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems.*, Theor. Comput. Sci. **165** (1996), pp. 97–131.

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

[13] Middeldorp, A., *A sufficient condition for the termination of the direct sum of term rewriting systems*, in: *LICS* (1989), pp. 396–401.

[14] Middeldorp, A. and Y. Toyama, *Completeness of combinations of constructor systems.*, in: R. V. Book, editor, *RTA*, Lecture Notes in Computer Science **488** (1991), pp. 188–199.

[15] Moses, T., *Extensible access control markup language (xacml) version 2.0*, Technical report, OASIS (2005).

[16] Pavlich-Mariscal, J. A., L. Michel and S. A. Demurjian, *A formal enforcement framework for role-based access control using aspect-oriented programming.*, in: L. C. Briand and C. Williams, editors, *MoDELS*, Lecture Notes in Computer Science **3713** (2005), pp. 537–552.

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

[18] Rusinowitch, M., S. Stratulat and F. Klay, *Mechanical verification of an ideal incremental abr conformance algorithm.*, J. Autom. Reasoning **30** (2003), pp. 53–177.

[19] Schneider, F. B., *Enforceable security policies*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.

[20] Toyama, Y., *Counterexamples to termination for the direct sum of term rewriting systems*, Information Processing Letters **25** (1986), pp. 141–143.

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