A policy-oriented language for expressing security specifications

by

Carlos Ribeiro and Paulo Ferreira

{Carlos.Ribeiro,Paulo.Ferreira}@inesc-id.pt

IST / INESC

Rua Alves Redol N^{o}9 1000 Lisboa

Portugal

Contact

Author: Carlos Ribeiro

Address: Rua Alves Redol N^{o}9 1000 Lisboa Portugal

Phone: +351 213100292

Fax: +351 213145843

email: Carlos.Ribeiro@inesc.pt

Abstract

Organizations' authorization policies are usually described by access control rules enforced on each protected object scattered all over the organization. Having a single global security policy specification would promote both security clarity and coherency [4, 9, 18, 31, 37].

Having a single security model for the whole organization, a single point of management and enforcement with a innumerous set of unknown users, does not scale well. However, both the policy enforcement and the mapping of unknown users to known entities [28] can be decoupled from the specification, thus having a single global security policy decoupled from the enforcement and from the mapping of unknown users promotes clarity and coherency without compromising scalability. This work presents a security policy language which is able to express simultaneously many different types of models, which is essential to handle all the policies used on a complex organization.

The proposed language can express the concepts of permission and prohibition, and some restricted forms of obligation. We show how to express and implement obligation using the transaction concept. We also address the problem of incoherent policies and show how to efficiently enforce the security policies expressed by the language with a security access monitor, implemented in java, including history-based and obligation-based security policies.

1 Introduction

One of the problems security administrators of organizations face is the lack of a global security policy specification. Organizations security policies are often described by access control rules (ACLs or other forms of simple rules) enforced on each protected object scattered all over the organization. This makes it very difficult for the organization to build a complete view of the policy enforced and makes it impossible to maintain coherency, thus undermining the trust in the system security.

Organizations with MAC policies are less affected by this problem but they suffer from the lack of expressiveness of such policies. Because of such lack of expressiveness, MAC policies are never used alone, and therefore the problem remains.

Having a single security model for the whole organization may not be appropriate at all. Moreover, having a single service, enforcing one or more closed models at the same time, does not scale in terms of efficiency and is not suitable for situations where users are not known in advance (*open world assumption*).

However, the enforcement and mapping of users to policies [28] can be decoupled from the actual policy specification. Therefore, there is a need for a global specification: (i) with enough expressive power to describe the different types of policies required by the departments and users of organizations systems, (ii) which can be globally checked for incoherencies, (iii) which is enforceable by the underlying applications and systems, and (iv) scalable, with the policy size, in terms of design, management and efficiency.

We have designed a security policy language (SPL) [41] comprising a small number of generic primitives, which is able to simultaneously express many different types of security policies within a single specification. We have also described a tool to automatically check the coherency of policies, and a compiler to enforce the policy with a security monitor. The use of a security monitor simplifies the deployment in most systems because it is the most frequent method of enforcing policies within applications and systems. SPL is currently being used within the Heimdal [24] architecture to implement security policies on grid platforms. The current implementation uses a centralized approach for policy evaluation. However, SPL can be used in a similar way to the STRONGMAN architecture [34], which implements the “distributed firewall” concept [3] using a KeyNote [10] library on every participating node.

SPL is composed of rules, policies and an algebra for rules. Rules decide whose actions are allowed and whose are denied. Policies provide encapsulation, parameterization, and inheritance, and the algebra provides the necessary composition flexibility to express many different types of policies. Among others, SPL is able to express the classic MAC, DAC, RBAC and delegation-based policies. It is also able to express obligation-based and history-based policies, which are essential to express current organizations' security policies.

History-based separation of duty is described as the most flexible form of separation of duty [48]. Obligation is a very powerful concept to express security policies [18]. However, neither history-based or obligation-based policies are easily enforced with a security monitor. The first ones because they have a log scalability problem and the second ones because they imply dependencies in the future [46]. We have shown that history-based policies can be efficiently enforced using a simple optimization algorithm and that, by using the transaction concept, an access control service based in SPL may enforce some forms of obligation.

The remainder of the paper is organized as follows. The next section presents some related work. Section 3 presents the SPL structure and basic blocks (rules, entities, sets and policies) and describes how to express and enforce policies with those blocks. Section 4 shows how to express and enforce history and obligation policies. Section 5 describes how SPL handles conflicts. Finally, in Section 6 the paper is concluded.

2 Related work

Traditionally, access control policies are described by their implementation (ACLs, Unix protection bits, or database table permissions). This technique is not suitable for organizations with more than a few computers and certainly not suitable for large organizations with several domains, neither in terms of management or security. To cope with the management and scalability problems, recent commercial solutions have adopted the RBAC model [29, 35]. However, several of these solutions are still not good enough in terms of expressiveness; namely, when it is necessary to express more flexible forms of separation of duty [22].

To overcome this lack of expressiveness several solutions have been proposed. Most of these solutions are language-based, but some are graphic-based and others use a language to complement an RBAC structure.

Some of the solutions use logic-based languages. Jajodia *et al* [31] define a stratified first-order language

with ten predicate symbols and three stratification levels. This language is able to express several different types of security policies at the same time, including history-based policies, and uses stratification to solve conflicts between policies (the conflicting policies are prioritized by rules at a higher level). Bertino [5] uses a different prioritization strategy to handle conflicts between rules. In [5] rules are prioritized based on their specificity, authorship, or modality (e.g. negative rules have precedence over positive ones). In SPL, conflicts are implicitly handled by the composition algebra. The policy architect chooses the prioritizing scheme when he composes different eventually conflicting rules.

Although modal conflicts are the most important type of inconsistencies, SPL also handles other types of inconsistencies which result in abnormal policy behavior. We have identified several types of inconsistencies within the security policy and between the security policy and a workflow engine. Ioannidis [30] identifies and solves some other types of inconsistencies. These inconsistencies occur when organizations apply a uniform global policy to a heterogeneous distributed system without a uniform namespace. Due to different meanings given to actors and resources, each system may end up implementing a different policy. SPL assumes a global uniform namespace similar to the one described in [42].

Woo and Lam [51] have shown how default logic can be used to express authorization rules. This logic is very powerful for relating rules to each other, which results in very expressive policies. Bertino *et al* [4] focus on the problem of expressing common temporal constraints (e.g. a resource that may only be accessed on Tuesdays) with a simple yet flexible language. Cuppens and Saurel [17] define a language for access control based on deontic logic. Deontic logic includes both the common permission and prohibition concepts together with the obligation concept, thus allowing the expression of obligation-based polices. Obligation can also be expressed in LGI [37].

LGI uses a distributed enforcement approach and uses a Prolog like language to define policies. It is able to express several types of policies including obligation-based and history-based policies. In LGI each participating entity engages in a group of entities governed by the same policy (called Law in LGI). The global group policy is locally enforced by each entity, thus ensuring event scalability. SPL may also be used this way; in particular, Dias's [19] shows how SPL can be used on an agent framework where each agent has

its own policy to enforce, although SPL by itself does not ensure event scalability. LGI enforces history-based policies by explicitly writing the code that creates the log. Instead, SPL infers that code from the policy itself.

Each logic-based language has its specific merits and problems. However, they all share the design scalability problem due to their lack of structure. All of them are composed of a list of rules which is likely to become too long to manage and is hardly suitable for reuse. This is the case of the Adage language [9]. The Adage language is composed of rules with a domain of applicability and a domain of acceptability. Adage is able to express many different types of policies including RBAC and history-based policies although it cannot efficiently implement the history-based ones [53]. Adage also provides a graphical interface to design and manage policies. This interface can minimize the design scalability problems but it was designed for "intermittent users" with less expressiveness requirements; thus, it is not suitable for designing large complex policies. This problem of expressiveness also exists in Lasco [33], which is another graphical solution for expressing security policies.

Some access control languages [1, 15] were designed to specify constraints over an RBAC model [45], thus increasing its expressiveness. RBAC is a very successful model that proved to meet reasonably large organizations' needs. The result of the combination of the RBAC structure with a constraint language is a more expressive model than RBAC without constraints, which meets large organizations' needs. Although more expressive than the original RBAC model, these languages were neither able to express history-based policies nor delegation. These limitations were addressed in [16] and [52], respectively.

Crampton [16] describes a history-based enforcement solution which is similar to the one followed in SPL. Both solutions try to solve the history log size problem by keeping just the information needed by active policies in the log. However, our solution is more general because SPL rules are more expressive than the policy tuples used in Crampton's work (e.g. SPL is able to decide based on the cardinality of previous accesses). Moreover, in an attempt to further reduce the size and improve the query efficiency of the log, the solution described in [16] divides the log among users, i.e. each user records and queries its own log; this means that a user cannot be prevented from doing something based on someone else's actions. In SPL, the log is also divided with the same goal, but it is divided among policies, and not among users, which is a better

partitioning solution because, by definition, each log will only be read by the policy requiring it.[1] Selectively writing events to logs is frequent in audit logs. For instance, Windows audit ACEs [49] define which actions should be logged. However, because their purpose is different, the log is not split for query improvement and, more importantly, the logged actions are explicitly defined by ACEs. They are not inferred from the access control policy as in SPL.

Delegation is a very important action for access control in distributed systems. In fact, delegation is the key principle of *trust management* systems [11, 10]. These systems use credentials, which are signed certificates with policies fragments or simple attributes [21], to specify the rights that belong to someone or to a key. However, these rights are only granted if the signer has himself credentials to do it. The final result is a global security policy comprising a distributed web of credentials. However, having a web of credentials specifying the security policy may not be acceptable to all organizations. Some of them may want to have a single and uniform view of its security policy. A similar solution is proposed in [28], but instead of using the web of credentials to define the policy, it uses the credentials to define a mapping between key holders and roles and leaves the definition of the policy to a classical RBAC system. This solution is particularly interesting because it decouples user-mapping from policy specification, thus allowing the use of any policy specification on systems where not all the users are known in advance (open world assumption).

Ponder [18] and XACML [26] are languages that allow the definition of organization-wide security policies. They share the characteristic of organizing rules into encapsulating units known as policies. XACML is a XML specification for expressing access control policies. The specification is able to express many different types of policies including RBAC, obligation-based policies and delegation-based policies. Although XACML rules are grouped into policies, policies cannot be grouped into other policies limiting the design scalability.

Ponder is a language that is very similar to SPL. It allows the definition of template policies that can be used to build policies easily. It is able to express RBAC, obligation, delegation and refrain policies,

[1] The problem with this solution is the eventual duplication of records in several logs due to the fact that a single action may have to be recorded in several logs. However, as shown in section 4.1.2, the information kept for each action in each log is very small and probably different.

where refrain policies are defined as negative obligations. The Ponder agglomeration strategy is supported by domains, which is a concept very similar to a directory which may contain files or other directories. The good thing about domains is that policies may be applied to every object deep inside a tree of domains. As far as we know, Ponder is not able to express history-based policies thus it cannot enforce history-based separation of duty as defined in [48]. The other main difference from SPL is that Ponder does not use a policy composition language. The absence of such a language forces Ponder to have several types of *composite* policies - group, role, rel and mstruct - for different usages. In SPL these *composites* are all performed by one entity - the policy. This solution is more flexible because it allows the addition of other types of *composite* without changing the language.

Some systems avoid the need for a composition algebra by partioning the set of protected objects and applying only one policy to each partition [6]. Although this a simple solution, it cannot take advantage of the flexibility provided by an algebra which is able to create a new policy through the combination of others, possibly already defined in a policy library [12, 50]. Bonatti [12] describes an algebra for composing policies expressed in different languages. The set of operators defined by SPL is similar to the one defined in [12] plus the quantification operators. In SPL the quantification operators are very important because they provide the ability to express both history-based and obligation-based policies.

Wijesekera [50] describes a different approach for expressing policies. They are not described as sets of permissions and prohibitions, but as transformations of authorizations. These policies decide which individual permissions and prohibitions should be added to the global set of permissions and prohibitions based on the activation of a set of rules. These rules are activated whenever the conditions described in each rule are met. These conditions can be the existence of specific authorizations or context information in the global set. Although it is very powerful, this approach has two problems. First, it does not provide a mechanism to group rules into generic policies; second, the number of facts (authorizations, permissions and context information) in the global set may grow to a very large number, degrading efficiency.

In conclusion, some of the most recent access control frameworks are very expressive (the logical based languages) but do not scale well in terms of design. Others, scale well in terms of design (RBAC solutions)

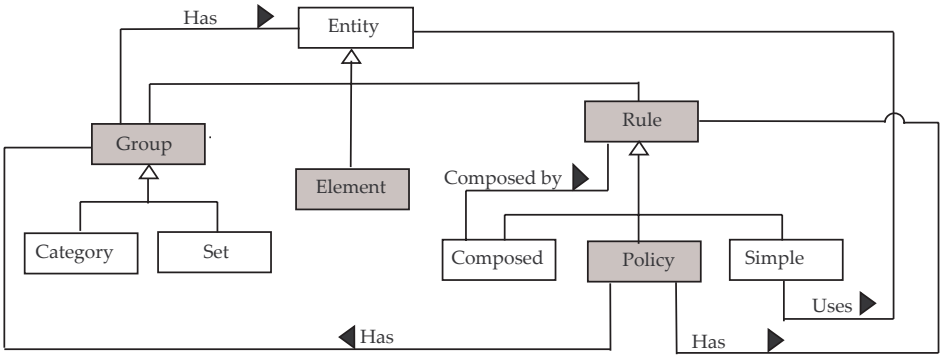

Figure 1: SPL entity relationship.

but are limited to one model (RBAC). Yet others scale well in terms of design but cannot enforce history-based or obligation-based policies. SPL builds all these proposals up into a solution that is able to express most policies described in previous systems. This solution scales better than most of the others in terms of design; is able to efficiently enforce history-based policies; and proposes a passive solution to enforce some obligation types.

3 SPL structure & basic blocks

In this section we present each one of the entities comprising SPL in detail and show how they are used in writing SPL security policies.

SPL is a policy-oriented constraint-based language. It is composed of four entities: elements, groups, rules and policies (Figure 1). The fundamental entity of the language is the rule. Rules express constraints in terms of relations between elements and groups. Policies are complex constraints that result from the composition of rules and groups into logical units. A complete BNF specification can be found in [38].

3.1 Elements

SPL elements are strongly typed entities with an explicit interface through which their properties can be queried. Every SPL element is a proxy to an entity of the underlying platform, thus enabling access to context information[2].

SPL elements may represent many different types, each one with its specific properties depending on the

[2]This solution is also found in Ponder [18] and in XACML [26], which use domains mapped to LDAP directories and XACML Contexts, respectively, and is opposite to the AAA IETF framework [2] in which the context information is provided by the application requiring the access.

target platform. For instance, in a file system the elements are files, subjects and access requests (from now on referred to as requests) over subjects and files. In a router, the elements are networks, hosts and messages, each of these having different properties. Each property may be a reference to another element, to a group or to one of the basic types `number`, `string` and `boolean`.

In many SPL target platforms, the SPL element set may form a polymorphic hierarchy, where each element is a specialization of another element. Figure 2 shows the element type hierarchy used in the examples in the next sections. The root of this hierarchy is the "element" type. The remaining element types are defined through the specialization of this base type.

One of SPL's strengths is its ability to be used in several target platforms. In the remainder of this paper, we use the very simple framework described in Figure 2. This framework has some properties that others may not have, that may prevent, or at least make the expression of some policies more difficult. For instance, in this framework, subjects are a specialization of elements. Every element has a creator, and every element knows which group it belongs to, which might not be the case of every framework. In fact, most access control systems have different hierarchies for targets and subjects. However, in some frameworks, subjects may sometimes behave as targets [19] and the framework should reflect that.

Although the ability to adapt to different frameworks is one of the strengths of SPL, it has some drawbacks. Because the framework used in each particular situation depends on the properties of the target platform chosen, the expressivity of the overall system is not the same in every platform. Thus, if Bertino's comparison framework [7] was used to compare the Bell-Lapadula model with SPL, we would achieve different results depending on the supporting framework.

```
type object {
  string name;        // The name of the object.
  user owner;         // The owner of the object.
  string type;        // A string identifying the type.
  object group groups;// The groups with the object.
  string homeHost;    // The host where the user
}                     // is defined.

type user extends object {
  rule group userPolicy; // user private policies.
}

type operation extends object {
  number ID;       // operation Id.
}
type event extends object {
  user author;     // The author of the event.
  object target;   // The target of the event.
  operation action;// The performed action.
  object group par;// The group of parameters.
  number time;     // The time instant.
  object task;     // The task to which
}                  // the event belongs to.
```

Figure 2: Example of an object type hierarchy definition.

3.2 Groups

Elements can be divided into groups. Groups are essential in any policy as they provide the necessary abstraction to achieve compactness, generalization and scalability. Without groups, each rule would have to be repeated for each element to which the rule applies.

```
external string localhost;    // An external entity
external user group AllUsers; // All the users
                              // in the system
external object group AllObjects; // All the objects
external operation group AllActions; // All the actions
external event group AllEvents; // All the events,
                                // past and future

                        (a)
```

```
// A category of all users that are
// defined locally
user group localUsers =
                AllUsers@{.homeHost = localhost };

// A group defined as empty
user group ActiveGroup = {};

                                                (b)
```

Figure 3: Example of: (a) external entities and sets; (b) a category and a group.

Groups can be internal or external. Internal groups are internal SPL structures with references to the entities contained in the group. External groups are proxies for groups in the target platform. Some external groups are very useful for the definition of policies; for instance, the groups of all subjects and all elements known to the system (Figure 3a).

SPL supports two types of groups: categories and sets. Categories are groups defined through the classification of entities according to their properties, e.g. all authenticated subjects in machine A; sets are groups defined through the explicit insertion and removal of their elements. The insertion and the removal of members from a set can only take place through external actions since SPL should not perform operations on external or internal entities that result in changes of state; this means that sets defined within SPL are managed by external tools. Both categories and sets are declared as groups, but are instantiated differently.

Categories are defined by restricting the members of other groups to the ones with particular properties. This is done by the SPL restriction operator (*mygroup@*{*logical-expression*}), which is a polymorphic operator that can be used in any type of group or rule (Figure 3b)[3]. The restriction operator has two operands, one is the group that it wants to restrict, and the other is a logical expression that must be satisfied by the elements in the group in order to belong to the restricted group. The logical expression uses properties of the entities in the group to define which members are selected. These properties are written with a dot before the name.

[3]See also section 3.3, for restriction on rules.

SPL defines five more group operators: the index operator ($mygroup[nth]$) that applied to a group returns the nth member of the group; the membership operator (*entity IN mygroup*); the cardinal operator ($\#mygroup$) that returns the number of members of the group; the union operator ($mygroup1 + mygroup2$); and the intersection operator ($mygroup1 * mygroup2$).

3.3 Constraint rules

SPL is a constraint-based language. Constraint languages are widely used to express systems and access control policies [9].

The language is composed of individual rules, which are logical expressions that can take three values: `allow`, `deny`, and `notapply`. Their goal is to decide on the acceptability of each request under the control of the access control service that implements the language. To make this decision, rules have an implicit parameter that represents the request upon which the rule decides. To distinguish this request from past and future requests we call it current request, and refer to it as `cr` .

A rule can be simple or composed. A simple rule is composed of two logical binary expressions: one to establish the domain of applicability and another to decide on the acceptability of the request. If the applicability expression evaluates to false the rule evaluates to `notapply`. If both applicability and acceptability expressions evaluate to true, then the rule evaluates to `allow`, and if the applicability expression evaluates to true and the acceptability expression to false, the acceptability expression will evaluate to `deny`.

```
[label :]  domain-expression ::  decide-expression
```

(a)

```
// Every event on an object owned by the author of the event is allowed
OwnerRule:  ce.target.owner = ce.author ::  true;

// Payment order approvals cannot be done by the owner of payment order
DutySep:  ce.target.type="paymentOrder" & ce.action.name="approve" ::  ce.author != ce.target.owner;
```

(b)

Figure 4: Simple rule: (a) syntax; (b) examples.

The SPL syntax for a simple rule (Figure 4a) has two parts: an optional label and two logic expressions separated by a special marker ('::'), representing the domain-expression and the decide-expression respectively.

The domain-expression and the decide-expression are simple binary expressions which use: the '$\&$', '|'

and '∼' logic operators, respectively for the conjunction, disjunction and negation; the '=', '!=', '<', '>', '>=', '=<' equality/inequality operators; and the "true" and "false" special values.

The domain-decide construction should not be considered a simple binary implication. If a binary implication was used, every rule would be implicitly open, i.e. every request outside the domain would be allowed, which is contrary to the SPL design principle of being able to express several different models simultaneously.

Figure 4b shows three simple rules, labelled `OwnerRule`, `DutySep` and `ClosedOwnerRule` respectively. The first one states that requests acting on a target owned by the author of the request (`cr.target.creator=cr.author`) is always allowed (decide-expression always true). The second rule states that payment order approvals are only allowed if the author is not the owner of the payment order. The third rule is similar to the first one, it allows the same requests, but it denies every request on a target that is not owned by the author of the request (whereas the first rule does not decide on those requests). The first rule is more appropriate for composition with other policies because it only decides upon its domain of applicability.

The domain-decide type of construction described above is simple, although it is more powerful than the permission and prohibition construction [31], in which each rule is exclusively a permission or a prohibition. Usually a permission/prohibition rule is composed of a domain-expression to identify which requests are allowed/denied and a keyword specifying the type of rule: permission or prohibition. The domain-decide construction is an extension of this permission/prohibition construction, in which the keyword specifying the type is replaced by an expression, thus allowing a rule to simultaneously express permissions and prohibitions.

3.3.1 Rule composition

In most rule-based authorization systems [31, 18, 26, 47], rules are combined into policies through an implicit conjunction of deny rules and an implicit disjunction of allow rules - each access is only allowed if none of the rules deny it and at least one rule allows it. With this type of composition it is not possible to specify open policies, only closed ones. Closed policies which deny every action not explicitly allowed, are the most frequent ones. However, in some cases, an open policy which allows every action not explicitly denied, is more appropriate [32].

In SPL, a rule can be composed of other rules through a specific tri-value algebra with three logic op-

erators: conjunction (AND), disjunction (OR), and negation (NOT). These operators behave as if their binary homonyms were applied to the decide-expressions of every rule applicable to each request. So, to evaluate the conjunction, disjunction or negation of rules, there are three steps: i) the applicability expression of each rule is evaluated for a specific request, ii) the rules inapplicable to that request are removed, and finally iii) the result of each decide-expression is combined using a binary conjunction, disjunction or negation, respectively.

```
// DutySep has a higher priority than OwnerRule
DutySep OR (DutySep AND OwnerRule);

deny:  true ::  false; // Implicit deny rule.
allow:  true ::  true; // Implicit allow rule.

// Simple rule conjunction,
//with default allow value
OwnerRule AND DutySep AND allow;
```

```
//User rules are restricted to their own objects
userPolicy:  FORALL u IN AllUsers {
              FORALL r IN u.userPolicy {
               r @ { ce.target.owner = u }
              }};

// Simple rule disjunction, with default deny value
// implementing a simple DAC policy.
DAC: OwnerRule OR userPolicy OR deny;
```

Figure 5: The Composition of rules using the conjunction, disjunction and restriction operators.

This tri-value logic allows some interesting constructions for access control expressiveness. For instance, a default value can be expressed using special rules in which the domain-expression is always true and the decide-expression is true or false depending on the default value (`allow` or `deny`, in conjunctions/disjunctions) as shown in Figure 5. Another interesting construction presented in Figure 5, shows how to express priorities between rules. The result of the `defaultValue` rule in Figure 5 is the result of the `DutySep` rule, except when this rule is inapplicable, in which case the result is equal to the result of `OwnerRule`.

Rules do not have to be written at the same time by the same author. In fact, they are usually dynamically written by several authors. It is often necessary to restrict the domain of applicability of a previously written rule, by the same author or by a different one, without completely removing it. For instance, a rule may state that the rules inserted and managed by each user can only apply to targets belonging to them. In SPL, this is achieved through the application of the polymorphic restriction operator (presented in Section 3.2) to rules and policies, in order to restrict their domain of applicability. Figure 5 shows how the restriction operator can be used to restrict the policies defined by each user to targets owned by him (the `userPolicy` rule), which together with the `ownerRule` can be used to define a DAC policy.

SPL does not provide a specific mechanism for delegation. Instead, it relies on the ability of each user to dynamically insert rules and policies into groups of rules. With these groups and the restriction operator, it is

```
// Who can delegate
who:  ce.target = dynamicRules & ce.action IN insertRemoveActions ::  ce.author IN Delegators;

// What can be delegated
what:  FORALL r IN dynamicRules { r@{.author IN Delegatees & .action IN AllowedActions} };

delegation:  who AND what;
```

Figure 6: Rules to control delegation of rights.

possible to specifically define who can delegate what to who (Figure 6). The `who` rule states that the insertion and removal of rules from the `dynamicRules` set can only be done by users belonging to the `Delegators` set. The `what` rule enforces every rule in the `dynamicRules` set, restricted to a set of actions and a set of delegatees, i.e. delegators cannot delegate everything to everyone.

SPL is also able to define the depth and width of a delegation [52]. Both depth and width can be tracked by history-based policies. With history-based policies it is possible to prevent a delegated right to be further delegated if there is a delegation chain which is bigger than a value (depth), or to prevent a right to be delegated by an entity more times than a specific value (width).

3.3.2 Quantifiers

In order to increase the flexibility of the composition, SPL defines three types of quantifiers over rules (Figure 7): one for each operator over rules - Universal quantifier (conjunction), Existential quantifier (disjunction) and Restriction quantifier (restriction). The Universal quantifier is defined as the tri-value conjunction of every instantiation of a rule over a specified set. The Existential quantifier has three qualifiers: AT_LEAST n, AT_MOST n and EXACTLY n, with the usual meanings. With the AT_LEAST qualifier the existential quantifier requires that at least n instantiations of the rule allow the request; with the AT_MOST qualifier it requires that at least one and no more than n instantiations of the rule allow the request; with the EXACTLY qualifier it requires that no more and no less than n instantiations allow the request.

```
// Universal quantifier syntax                          // Restriction quantifier
FORALL var IN set { rule_skeleton(var) }                rule@{ expression(var) } WITH var IN set

// Existential quantifier syntax
EXIST AT_LEAST n var IN set { rule_skeleton(var) }
```

Figure 7: Universal, existential and restriction quantifiers syntax.

The Restriction quantifier is slightly different from the first two and complements them. While the universal and existential quantifiers operate over the decide-expressions of rules (remember that tri-value con-

junctions and disjunctions are translated to binary conjunctions and disjunctions over the decide-expressions of rules), the restriction quantifier operates over the domain expression of rules. It restricts the domain of applicability of a rule to a domain that is defined by a universally quantified parameterized expression over a specified set.

3.4 Policies

An SPL policy is a collection of rules and groups that govern a particular domain of requests. Each policy has one "Query Rule" (QR) (identified by a question mark before the name of the rule) that relates all the rules specified in the policy. This rule uses the algebra defined before to specify which rules should be enforced and how. The domain of applicability of a policy is the domain of applicability of the QR.

Unlike several logical based languages, in SPL there is not an implicit operation between rules, i.e. the rules inside a policy do not form an implicit disjunction or conjunction. The expression formed by the rules is given by the query rule. This solution provides flexibility to the language because the user building the language may choose whatever construction is the best. In fact, he may even choose to extend a predefined policy (see inheritance below in this section) which has a predefined query rule that performs the conjunction or the disjunction of every rule in the policy.

In an SPL policy some of the groups can be parameters that are passed to the policy whenever it is instantiated (or, more correctly, activated). This allows for the construction of several abstract policies, which may be activated several times with different parameters. For instance, it is possible to have a generic DAC policy, a generic separation of duty policy, or a simple generic ACL policy (Figure 8a).

When instantiated, a policy acts as a rule and can be included in another policy by composing it with other rules through the tri-value algebra. As in several object-oriented languages, instantiation is performed by the `new` keyword. Figure 8b shows a security policy (`InvoiceManag`) that activates an ACE policy and delegates the decision on request acceptability to it.

The ability to merge policies into more complex ones, using the tri-value algebra, is one of the important features of SPL because it allows for the development of libraries of frequently used security policies. These security policies can then be used as building blocks for more complex security policies, thus simplifying the

```
policy ACL(
 user group AllowUsers, // Users that are allowed to
                        // perform restricted actions
 object group ProtObjects,  // The protected objects
 interface RestrictActions) // The restricted actions
{

?Psimple:
 ce.action IN RestrictActions & // if event action
                                // is restricted
 ce.target IN ProtObjects // and target object
                          // is protected then
 ::ce.author IN AllowUsers// the event is allowed
                          // if the author is allowed
}
```

(a)

```
policy InvoiceManag
{
 // Clerks would usually be a role
 // but for simplicity here it is a group
 user group clerks ;

 // Invoices are all objects of type invoice
 object group invoices =
              AllObjects@{ .doctype = "invoice" };

 // In this simple policy clerks can
 // perform every action on invoices
 DoInvoices:  new ACL(clerks, invoices,
              AllActions);
 ?InvoiceManag:  DoInvoices;
}
```

(b)

Figure 8: Definition and instantiation of a generic policy: (a) A generic ACL policy, with three parameters enclosed in parenthesis next to the name; (b) a policy instantiating the ACL policy.

specification of security policies for complex organizations.

The natural SPL policy sharing mechanism is delegation, but SPL also supports policy inheritance to simplify some sharing situations. For example, defining a policy which is similar to another policy with only one slightly different rule is much more difficult with delegation than with inheritance.

```
policy genericRole (user group Authorized,
                    user group Active)
{
 // Events inserting a user into
 // the Active group are allowed only if
 // that user is in the Authorized group
 ?genericRole:  ce.action.name = "insert" &
                ce.target = Active
              ::  ce.par[1] IN Authorized ;
}
```

(a)

```
policy clerkRole (user group Authorized,
user group Active) extends genericRole
{
// Invoices are all objects of type invoice
object group Invoices =
             AllObjects@{.doctype="invoice"};
// All Active group members may access Invoices
Invoice:  new ACL(Active, Invoices, AllActions);

?clerkRole:  ?super AND Invoice;
}
```

(b)

Figure 9: SPL roles: (a) a generic role policy with no rights; (b) an example of an effective role policy.

The use of inheritance is particularly interesting for specifying RBAC policies. In SPL, roles can be defined as policies comprising the group of subjects allowed to play the role, the group of subjects playing the role and the set of rules that specify the rights of the role (Figure 9). This solution is very similar to the one proposed by Lupu and Sloman [36] in which roles are groups of rules. However, while in Lupu's work [36] roles are special groups of rules, containing special properties, in SPL roles are just another template policy. Figure 9 contains two policies: the `genericRole` policy and the `clerkRole` policy. The `genericRole` policy contains: i) the group with the subjects allowed to play the role, ii) the group of subjects playing the role and iii) a rule specifying that only the subjects in the first group can be inserted in the second. This

generic role does not contain any special right, only the basic role infrastructure. The `clerkRole` policy inherits these properties from the `genericRole` and adds rules specifying the permissions and prohibitions of clerks. As in other object oriented languages with single inheritance, the keyword `super` designates the inherited policy, and the keyword `?super` designates the query rule of the inherited policy.

SPL policies are only active if instantiated and inserted into another policy, except for the master policy, which is implicitly activated by the security monitor. The result is a hierarchical tree of active policies with the master policy on top. Only the master policy is queried. If the result is `deny` or `allow` the monitor should reject or accept the request, respectively. If the result is `notapply`, it is either an error that should be solved by the conflict verification tool (see Section 5) or it is used by the monitor to return information for user awareness and then accept or reject the request. This structure has several advantages over a flat one [9, 31, 51]. First, it clearly identifies which rules are related to each other, simplifying the global understanding of the policy. Second, it allows the dynamic activation and deactivation of policies, by inserting and removing them from other policies. Third, it partially solves the problem of conflicting policies (see Section 5).

3.5 Implementation

One of the problems of expressive security frameworks such as SPL, is the low efficiency of their implementations. While usual frameworks built upon access control lists, labels or Unix permission bits were designed to be efficient, SPL was designed to be expressive. In this section, we show that, using a mixture of compilation and query techniques, it is possible to achieve acceptable performance results, even for policies with thousands of rules. We have designed and implemented a compiler for SPL, which generates standard Java and is able to detect special SPL constructions and generate the most efficient code to implement them.

Given the resemblance between SPL and Java structures, most of the compiler's actions are simple translations:

- each SPL policy is directly translated into a Java class;
- each rule is translated into a tri-value function without parameters (with the exception of the query rule which has one parameter – the current request);

- each element is translated into a Java interface;
- each group variable is translated into a Java variable of the `SplGroup` type, which defines an interface to access several kinds of groups (external groups, subgroups of external groups, internal groups).

Wherever a policy instance is used on behalf of a rule, the compiler executes an automatic cast operation that consists in making the call to the query rule of the policy explicit. Thus, the overall structure of the generated code can be seen as a tree of tri-value functions calling other functions, in which the root is the function resulting from the translation of the query rule of the master policy and the leaves are the functions resulting from simple rules.

However, evaluating a tree of logical expressions for each request may pose scalability problems. While in standard ACL based systems only the access control entries (ACE) belonging to the ACL of each target are evaluated in each access, in SPL potentially every rule has to be evaluated for every access. This is a problem in systems with thousands of rules, subjects and targets.

Nevertheless, given that SPL is a logical-based language, it is possible to apply some evaluation optimizations. In a conjunction of rules (tri-value conjunction as defined in §3.3), if a rule evaluates to `deny` than it is not necessary to evaluate the remaining rules. This is similar for the disjunction of rules and `allow` values. Unfortunately, these optimizations are not very useful because the disjunction of rules are rare and the optimization applicable to conjunctions can only optimize the denial of requests.

Another useful optimization can be applied to the restriction operation (*rule@expression(request)*). The "restriction operation" restricts the domain of applicability of a rule to the set of requests satisfying a logical expression. Thus, if that expression evaluates to "false", it is not necessary to evaluate the rule. This optimization is very useful in those situations where rules are explicitly organized in domains of applicability (e.g. rules that only apply to targets produced by one branch of an organization). However, it is not enough to prevent the unnecessary evaluation of inapplicable rules inside the same domain. Whenever the restriction operation is not used to reach the conclusion that a branch of the evaluation tree is not applicable to a particular request, it is necessary to evaluate each domain expression of every leaf rule in that branch. A possible solution would be to build a virtual restriction operation in which the restriction expression would be the

logical disjunction of each domain expression[4] of every leaf rule in the branch. Although this solution is very efficient in detecting inapplicable branches, it penalizes applicable branches with the redundant evaluation of domain expressions in each node of the evaluation tree.

The solution used in SPL is based on the assumption that most rules are target-limited, in the sense that they are only applied to a limited set of targets. SPL is able to express non target-limited rules (e.g. all actions performed by a subject); nevertheless, we believe that most security policies expressed in SPL will be target-limited. This assumption is based on the observation that most current security policies are target-limited; e.g. all ACL based policies, Chinese wall policies, DAC and RBAC policies.

Based on this assumption we have designed a target-based index for rules, which allows for quick cuts on branches of the rule evaluation tree. The system creates an index for each target. Each index is maintained in the corresponding target as a label and keeps the information of every rule that may be applicable to a request with that target. The representation of that information in the current prototype is kept on a bit stream with one bit for each rule in the system. However, given the sparse nature of the information (we expect that only a few rules are applicable to each target as in current ACL based systems), it is possible to develop more compact structures.

This index technique has proved to be efficient, showing, on average, a speed-up of one order of magnitude. In particular, for a policy with 4120 rules, 12000 targets and 5000 subjects, it showed a speed-up of 4.8. The final time to evaluate the rules was around $5\mu s$ which is about 6% of the time to open a file for reading in the testing platform (Pentium M at 1.86Ghz running the Sun Java 1.5.0 virtual machine over Windows XP 5.1).

4 Special constraints

The previously described language can be used to express several types of constraints, including complex constraints that require special implementation considerations. In this section we show how to express and implement two special types of constraints with a request monitor: history-based constraints and obligation constraints.

[4]Obviously a reduced canonical form.

4.1 History constraints

Several security policies require requests to be recorded, in order to implement constraints with dependencies in the past. Among them, the Chinese Wall policy [13] is one of the best known. But many other forms of separation of duty [43] also require request recording.

The importance of history-based polices has been recognized by several authors [20, 43, 53, 31, 16, 37], however, no framework is able to simultaneously express concisely and implement efficiently every history-based policy that SPL does. Sandhu's *transaction control expressions* [43] are not able to express all types of history-based policies because not every action in long term targets is saved. Jajodia's work [31] writes every action in the global set of rules, which does not scale. The current version of Adage does not define an enforcement framework [53]. Deeds [20] and LGI [37] implement history-based policies directly in java and prolog, respectively, which requires specific programming for each history-based policy. Finally, Crampton's work [16] is similar to our own but our solution is more general because SPL rules are more expressive than the policy tuples used in Crampton's work (e.g. SPL is able to decide based upon the cardinality of previous accesses).

4.1.1 Expressing history constraints

In SPL, history-based policies are expressed by simple quantification rules over the abstract PAR (Previous Accepted Requests) set. Each of these rules declares and quantifies one variable, used to classify each type of past request monitored by the access monitor. Thus, to monitor a sequence of requests in SPL, it is necessary to cascade several quantification rules over the PAR set, one for each type of request. Figure 10 shows a Chinese wall policy with one class of interest expressed in SPL.

The Chinese Wall policy is a monotonic history-based security policy, designed for open systems, i.e. inserting the Chinese wall policy into a generic policy cannot increase the set of allowed requests. Briefly, the policy states that targets are divided into classes of conflicting interests. Each object has a owner and a subject can access every target, but in each class of interest he can only access the targets belonging to a single owner.

The policy in Figure 10 defines a group and a rule. The group contains all the targets with the same

conflict of interest. The rule states that the current request is denied if the target of the request is in the "interest class" and there is a past request (`pr`) performed by the same subject on a target with a different owner that belongs to that "interest class".

```
policy ChineseWall(element group InterestClass) {
?ChineseWall:
 EXIST pr IN PAR {
  cr.target IN InterestClass & pr.target IN InterestClass & cr.author = pr.author &
  cr.target.creator != pr.target.creator ::  false
 };
}
```

Figure 10: A specification for the Chinese Wall policy; `pr` stands for past request

Usually an organization implementing a Chinese Wall policy has several classes of conflicting interests. The above mentioned policy has just one class, but can be instantiated several times, one for each class of interest.

The decide-expression of the rule has a constant value, which is consistent with the monotonicity of the Chinese Wall definition. This definition specifies which requests should be denied, but leaves it up to complementary policies to decided upon the ones that should be accepted. If, for instance, the expression `cr.target.creator != pr.target.creator` is moved from the domain-expression to the decide-expression, the policy result could either be `allow` or `deny`, which is against the monotonicity of the policy definition.

4.1.2 Implementing history constraints

A monitor-like security service has to decide, for each request, whether it should allow or deny the request. The decision must be taken at the time of the request with the information available. Thus, in order to implement history-based policies, any monitor-like security service has to record information about past requests.

Some security services record requests implicitly in their own data structures [44] (mostly using labels), others record them explicitly into a request log [9, 31] that can later be queried for specific requests. The latter solution is more flexible than the former but if the request log becomes too big, the memory space required to keep that log may become unlimited and the time required to execute each query could have a significant impact on the performance of the system. Jajodia [31] tries to solve this problem recording the requests that

differ in time only once. However, this does not solve the problem because the number of requests to record is still huge and prevents policies based on request cardinality to be enforced, e.g. the user may only login three times in the system.

We show that it is possible to efficiently implement the log solution, both in terms of memory-space and performance. This is obtained through a compilation algorithm that optimizes the amount of information to be saved and the way that information should be queried. We show that, although this algorithm does not obtain the best results for all history-based policies, the results obtained for most frequent policies are equivalent to those obtained by label-based implementations [44].

The goal of this algorithm is three-fold. First, the security manager should selectively log just the requests required by the history-based policy, e.g. if a policy needs to know if a document was signed, there is no need to record requests that are not "sign requests". Second, the security manager should selectively log just the fields of the requests required by the specified history policies , e.g. if a policy wants to decide based on whether or not the author of the current request has signed a document, it is not necessary to record the "time" or the "task" fields of signature requests. Third, the security monitor should use the best possible structure to maintain the log and the best type of query to search it. The log is going to be searched by entries with specific properties. These properties might be expressed using equality constraints, inequality constraints or membership constraints. Equality constraints can be searched in a hash table in $O(1)$, which makes them ideal to be used as index keys. However, if there is not a single equality constraint to look for, it is better to use a balanced tree to hold the log and use a different type of query.

The main drawback of the proposed solution is that history-based policies cannot decide on requests prior to their activation, i.e. the system only records requests for each history-based policy after the policy starts to exist.

Instead of building a single log for every history-based policy, the compiler builds a specific and fined tuned log for each history-based policy. This solution has several advantages. First, it divides the problem reducing the number of requests required to be searched. Second, it allows for a better adaptation of the base structure to each query, because each log can be kept by a different structure. Third, it simplifies the

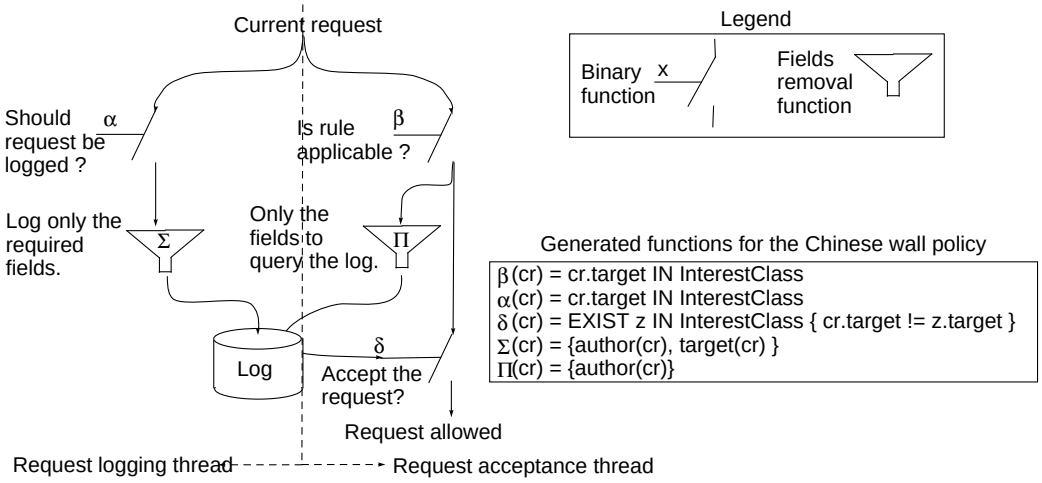

Figure 11: Schema representing the evaluation of a history-based rule.

insertion and the removal of policies. The problem in this solution is the potential for maintaining redundant information in several logs. However, given that the information kept by each log is the minimum information necessary to that policy, the level of redundancy expected is similar to the one of label-based implementations, where the labels used by different policies may also be redundant. Nevertheless, this problem can be further limited through the sharing of logs with the same signature (same requests to log, same fields in those requests to log, same base structure) between policies.

Figure 11 shows a graphic representation of the evaluation of a history-based rule for a request. The evaluation process is composed of the evaluation of a set of functions generated by the compiler. Some of those functions are binary functions which decide if the process should stop or proceed, others are field removal functions whose job is to strip requests from unnecessary fields.

The process is divided in two threads. The first thread decides if the request is denied or not. The second thread decides if the request should be logged or not. The β function decides if the rule is applicable to the request; if it is not, the acceptance thread stops[5]. The α function decides if the request should be logged for later use. The Σ function decides which fields of the request should be logged. The Π function decides which fields the log should be queried by. Finally, the δ function is the original policy stripped of all the expressions already evaluated. Together with the logged fields, there is also a counter with the number of times that each distinct tuple occurs, to avoid repetition of entries.

[5]In the Chinese wall policy this function verifies if the target of the request is in the interest class of the policy. If it is not, the policy does not apply.

With this solution, the data kept in the log is small and, more importantly it does not grow indefinitely. For instance, for the Chinese wall policy the number of entries in the log is, at most, equal to the number of subjects in the system. This is the direct of result the fact that the log does not have to keep repetitions and of the specific nature of the Chinese wall policy. The only past information needed to apply a Chinese wall policy with one class of interest is the identity of the authors that have accessed targets in that class of interest. This property is also common to all history-based policies that can be implemented with labels. In fact, the goal of the algorithm is to automatically find the information that a security designer would program to be saved in the labels on a label-based implementation.

Since the log size is bounded, the time needed to evaluate a history-based policy is also limited. In fact, this was confirmed by measurements taken in the current prototype. Figure 12a shows the time required to evaluate a Chinese wall policy with 10 classes of interest, according to the number of evaluated and potentially logged requests. As it can be seen, the time required remains constant confirming the expected behavior.

The time required to evaluate the policy is also not affected by the number of subjects or the number of targets in each class of interest. However, it is severely affected by the number of classes of interest (Figure 12b). This result is a direct consequence of the number of rules used to build the Chinese Wall policy with different numbers of classes of interest. The Chinese Wall defined in Figure 10 requires the definition of a rule for each class of interest. Thus, for Chinese Wall policies with more classes of interest, the monitor needs to evaluate more rules for each query.

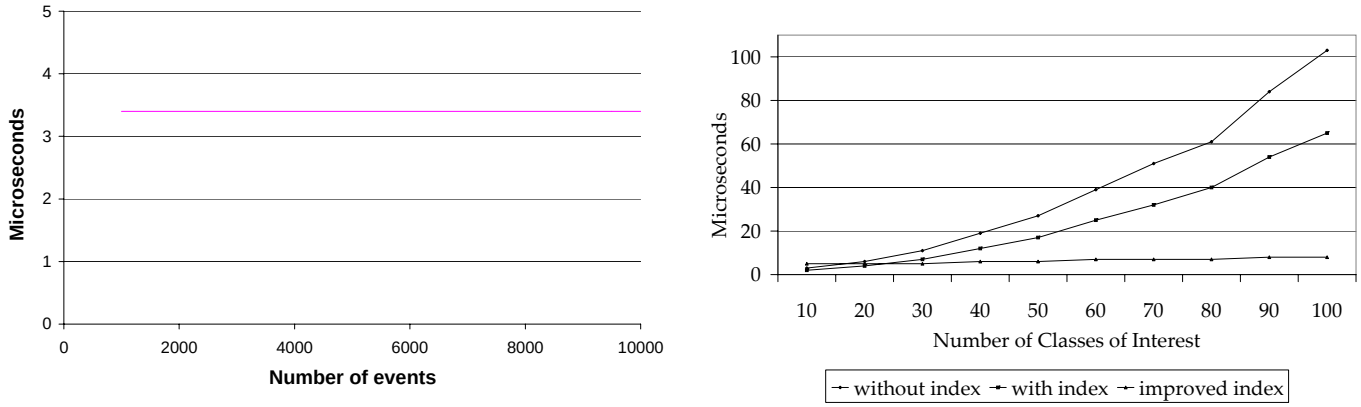

Figure 12: (a) Chinese Wall dependency with the number of events queried. (b) Chinese Wall scalability with the number of classes of interest.

The index solution presented in §3.5 can minimize the problem as shown in Figure 12b. However, it

is not enough for policies such as the Chinese Wall or any other policy with one single large conjunction of rules. In these policies, the index effectiveness is small because the branches in the evaluation tree of those policies are small. Thus, the cuts which the index is able to perform are necessarily small. These types of policies require better indexes, for instance, indexes with several layers of indexes over indexes. This solution is not implemented in the current prototype but its effect can be measured because it would be similar to rearranging the policies in order to have a deeper evaluation tree. For instance, the big conjunction of rules in the Chinese Wall policy can be rearranged into a conjunction of conjunctions using the associative property of conjunctions. The results in Figure 12b show the effectiveness of such approach.

4.2 Obligation constraints

SPL is able to express the concepts of permission, prohibition and obligation. While the first two are usually supported by classical access control services, the last one is not. However, several access control solutions have recently started to recognize the importance of obligations to express current security policies [17, 18, 26, 37].

Although most solutions are different from each other, all of them use some kind of special predicate to express obligations. This solution is simple and clear. However, with constraint-based languages the use of a special predicate to express obligation is an unnecessary addition.

Defining an obligation is not simple. In deontic logic, an obligation is defined as the prohibition of not doing something [14]. The problem with this definition, in a constraint-based system, is that rules decide upon requests, not upon users or other agents. Thus, a rule forbidding a request from not happening cannot be expressed. In SPL an obligation is defined as *a constraint with a dependency in future requests* [40]. This definition can easily be shown to be equivalent to a *triggered obligation*.

Triggered obligations are those obligations activated by triggering actions. These obligations can be represented by the following generic expression "*if do TriggerAction then must do ObligatoryAction*", which can be shown to be equivalent to "*cannot do TriggerAction if will not do ObligatoryAction*"[6], which is a constraint with a dependency in the future.

[6] $O \Leftarrow T \equiv \neg T \Leftarrow \neg O$

SPL specifies constraints with dependencies in the future the same way it expresses constraints with dependencies in the past, but with the special group PAR replaced by the special group FAR (Future Accepted Requests). Figure 13a shows a policy which specifies that when someone registers as a student of the "Online University", he is obliged to register as a student of at least one discipline from that university. This is achieved by constraining the action of registration in the University to the eventuality that in the future a request for action of registration in one of the disciplines will take place.

```
policy Registration(university OnlineUniv) {
 ?Registration:
  EXIST e IN FAR {
   cr.action.name = "Register" &
   cr.target = OnlineUniv ::
   e.action.name = "Register" &
   e.target IN OnlineUniv.disciplines;
  }
}
```

(a)

```
policy ModRegistration(university OnlineUniv) {
 ?ModifiedRegistration:
  FORALL tpr IN PAR {                    // Step 1
   EXIST e IN PAR {                      // Step 2
    cr.action.name = "commit" &          // Step 3
    cr.transaction = tpr.transaction &// Step 3
    tpr.time < e.time &                  // Step 2
    tpr.action.name = "Register" &       // Step 1
    tpr.target = OnlineUniv ::           // Step 1
    e.action.name = "Register" &
    e.target IN OnlineUniv.disciplines;
   }}}
```

(b)

Figure 13: An example of an obligation policy: (a) the Registration policy expressed as a constraint with a dependency in the future; (b) Registration policy after the aging process (the lines modified by each step of the process are marked).

4.2.1 Conditional Obligations

Not every statement expressing a conditional obligation in common language requires an obligation-based policy to be enforced. For instance, the statement"if someone executes some application he must register as a user" contains a conditional obligation but it can usually be expressed as a conditional prohibition with a dependency in the past: "Someone may execute an application if he has previously registered as a user". Thus it is necessary to clearly identify the situations where an obligation-based policy must be used.

We have identified two generic situations where an obligation-based policy is required. The first one occurs when the two actions involved in a conditional obligation, oblige each other. The situation described before, where someone is obliged to register as a student of at least one discipline if he has registered as a student of the *Online University* and vice versa, can be given as an example. In this situation, it is not possible to rewrite the statement as a conditional prohibition because, whatever action is performed first, there is always an obligation to fulfill.

The second situation occurs when the obligatory action is causally dependent on the trigger action. For

instance, if the obligatory action requires a value obtained by the trigger action, the obligatory action cannot be executed before the trigger action, forbidding the transformation of the conditional obligation into a conditional prohibition.

4.2.2 Implementing obligation constraints

Although the problem of expressing obligations with a constraint-based language can be solved by transforming them into constraints with dependencies in the future, they still cannot be easily implemented by a security monitor. In fact, Schneider [46] proved that constraints with dependencies in the future cannot be enforced by a security monitor. This is why most access control systems use some kind of engine to execute the obligatory actions [18, 26]. The problem with this solution is that the actions performed by the engine are not accountable to anyone and the engine is, itself, a single point of attack (whoever controls the engine can do everything).

In SPL, obligations are enforced by a passive security monitor without requiring a specific engine to execute the obligatory actions. In order to enforce constraints with dependencies in the future with a security monitor, SPL requires that both actions - the trigger action and the obligatory action - are executed inside an ACID[7] transaction. This allows the monitor to enforce an equivalent history-based policy, which prevents the commitment of the trigger action if, by the time of committing the transaction, the obligatory action has not taken place.

The process of translating a policy with a dependency in the future into a history-based policy is called *aging* [40]. The process has three steps. Each step adds or modifies some constraints (Figure 13b):

Step 1 takes any reference to variable `cr` (Current Request) and translates it to `tpr` (Trigger Past Request) which is universally quantified over PAR. If there are other variables quantified over PAR, it is necessary to add a new constraint specifying that `tpr` is the last of them.

Step 2 replaces each reference to the FAR group by a reference to PAR, and adds a new domain restriction to specify that `tpr` happens before any of them.

[7] ACID stands for: Atomicity, Consistency, Isolation, Durability

Step 3 adds two more domain restrictions, defining the request to be allowed or denied (current request). After the aging process, the request to be monitored is the request for the action of committing the transaction containing the trigger request action.

However, it is not always possible to perform both the trigger action and the obligatory action inside a transaction because some actions cannot be undone, e.g. sending a document to a printer or showing a text on the screen. These actions are called *real actions* on transaction management systems [27] and are already known to require special treatment by those systems in order to achieve atomicity. Usually systems delay the execution of such actions until all the other actions are executed, but if these actions cannot be reordered, the system is not able to ensure atomicity.

The problem is slightly more complex than in usual transaction management systems because the set of actions identified as *real actions* must include actions that change human knowledge state (e.g. showing a text on the screen), which are not often considered.

Therefore, both the usual active method and the proposed passive method of implementing obligations have problems and advantages. The active approach (the system executes actions by itself) is more flexible and is a broader solution. The passive approach (the system only denies or accepts requests) is useful for situations where a transaction monitor is available and there is a requirement for the accountability of every action.

5 Conflict resolution

SPL supports non-monotonic policies, in the sense that it is able to express both positive and negative constraints at the same time. The increased expressiveness with the addition of non-monotonicity does not come without cost as it leads to potential conflicts between contradictory rules. Usually these conflicts are solved with the introduction of implicit priority algorithms that choose which rule overrides the other. Some of these algorithms are very simple (e.g. negative rules override positive ones), others are more complex and use not only the rules' types but also the authority of the rules' issuers (i.e. rules issued by a higher authority manager override others), the specificity of the rules (more specific rules should often override more generic

ones), and the issuing time of the rules (more recent rules override older ones) [5].

Windows 2000 uses a simplified version of this approach. It evaluates ACEs in a special order and stops when it finds one which is applicable to the subject performing the request. The order in which ACEs are evaluated gives priority to non-inherited ACEs (priority to the more specific) and, within these, it gives priority to negative ones. This approach is very intuitive and natural, but it has some drawbacks. It is not unusual for a high authoritative manager to issue a rule which may be overridden by a low authoritative manager, or to express a mandatory general rule which should not be overridden.

Another strategy is to stratify the security rules and include a special layer of rules to decide which rules should override the others [8, 31]. SPL follows this strategy but, instead of creating a special layer of rules to solve conflicts, it forces the security administrator to combine policies into a unique structure which is, by definition, free of conflicts. In SPL, every active security policy must be in the hierarchical delegation tree of policies. Therefore, if two active policies present conflicting results to the same request (one denying it and the other allowing it), then somewhere up the hierarchical tree they must be combined in one tri-value expression that inherently solves the conflict. If the two policies are combined using a tri-value “AND”, the request is denied. If they are combined using a tri-value “OR” the request is allowed. With this solution, the policy architect is able to prioritize rules the way it pleases. The policy architect may choose a prioritization through specificity or through ownership or through any other prioritization method.

However, these mechanisms should not be used to solve real inconsistencies derived from the unification of several policies from several sources. In fact, they can even be detrimental, because they can mask real inconsistencies and produce wrong results. For instance, a senior system administrator can insert a rule which denies some accesses and later a junior system administrator, unaware of that rule, can insert one that specifically allows that type of request. Should we use the most specific, the most recent, or the rule issued by the most senior system administrator? The choice would probably be the most specific one, but what if the junior system administrator has made a mistake? An automatic conflict resolution mechanism would mask this error.

Although conflicts between contradictory policies are the most important type of inconsistency that may

be present in a global security policy, they are not the only ones. For instance, a policy may be completely overridden by another policy in such a way that the former policy is completely useless; or the combination of two or more policies may result in a policy that denies or allows every action in the system.

Furthermore, within an organization, it is not only necessary to verify the self-consistency of the security policy but also to verify the consistency of the security policy with other specifications of the organization. For instance, if a workflow application of an organization requires access to some documents and the security policy forbids that access, then the security policy is inconsistent with that workflow specification, which may prevent the organization from working as expected.

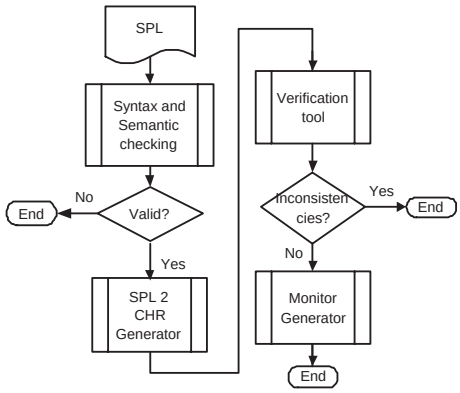

Figure 14: Overall flow of actions to generate a monitor from an SPL policy.

In SPL, these incoherencies are detected by a specific verification tool [39], which may run between the syntactic/semantic verification phase and the code generation phase of the SPL compiler (Figure 14). This tool is composed of: a generic constraint solver engine [23], a set of specific rules for SPL operations [38], and a translator from SPL to the specific logical language handled by the generic constraint solver [38].

The generic constraint solver solves constraints defined in the CHR (Constraint Handling Rules) language [23]. This language is composed of rules organized in handlers. Each set of rules solves a specific type of constraint. For instance, Figure 15a shows two of the rules used to solve order constraints (defined with the $<, >, =, \leq, \geq$ and $\neq$ operators). One of these rules is a *simplification* rule and the other is a *propagation* rule (denoted by the $\Rightarrow$ operator). Whenever applicable, the simplification rule replaces the constraints matching the head of the rule by the constraints matching the body of the rule. The propagation rule just adds the constraints to the body of the rule. Both rules are applicable when some of the constraints to be solved unify with the head and the guard is true. The successive activation of these rules gradually simplifies the constraints in

the goal and eventually finds a solution for the free variables in the goal, or reports an incoherency.

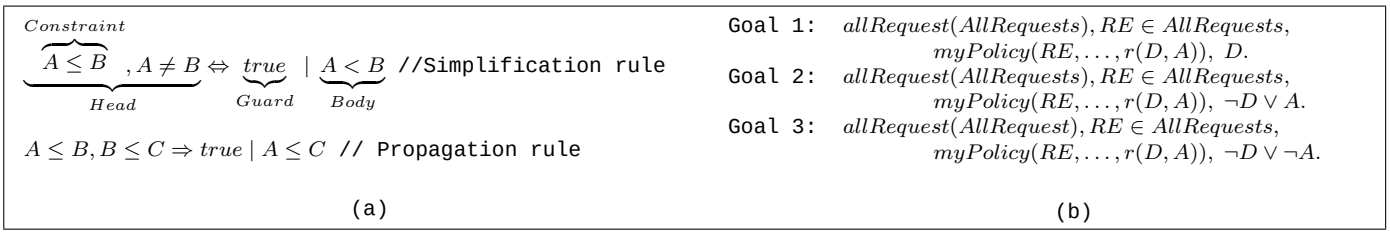

Figure 15: (a) Examples of the simplification and propagation rules. (b) Examples of incoherencies found by the tool. Goal 1 verifies if the policy is applicable to at least one request. Goal 2 and Goal 3 verify if the policy does not deny or allows every request, respectively.

To use this generic constraint solver engine to find incoherencies in SPL, it was necessary to write a program composed of user- defined CHR rules which is able to handle all the types of constraints that can be specified with SPL operators. The problem with this approach is the number of different rules required to cover all SPL operators and the scalability of the existing methods of proving correctness and termination in this type of program. To cope with this problem the program was divided into a set of *semi-independent* [38] groups of rules, named *handlers*, for which termination and correctness can be individually proved. It can be shown that the composition of a set of semi-independent, terminating, and correct groups of CHR rules, is also correct and terminates [38].

The tool is able to find many different types of incoherencies. To find each type of incoherency, the policy is solved with a specific goal. For instance, Figure 15b shows three different goals to detect three different types of policies. Each goal assumes that the predicate $myPolicy(RE, \ldots, r(D, A))$ defines a generic rule ($r(D, A)$) comprising every constraint of the policy. The first goal verifies if the policy is applicable to a request; if it is, the tool shows a solution for variable RE for which the domain of the policy (variable D) is true. The other two goals verify if the policy denies or allows every request.

The current prototype has ~ 300 rules and is able to solve all SPL constraints, including the constraints implicitly qualified with time. For instance, the SPL expression "cr.target IN GenericGroup" is implicitly qualified with the time of the request. Only at that time, the target must be in the group, thus when translating it to a CHR constraint, it must be qualified with a specific time. The complete set of rules and the proofs of termination and correctness of the tool can be found in [38].

6 Conclusion

We have defined an access control language that simultaneously supports multiple complex policies, and either has a higher expressive power, or presents better results in terms of design than other multi-policy environments. The language uses its hierarchical based policy-oriented structure to solve conflicts between simultaneously active policies. We also provide a tool to verify incoherencies within policies which goes beyond conflict detection.

The language was designed to be easily enforced by a security monitor. We have shown how index techniques can be applied to the policy structure to efficiently implement most security policies. Special care was taken with the enforcement of history-based policies. We have shown that by generating specific and special tuned logs for each history-based policy it is possible to implement SPL history-based policies as efficiently as handcoded label-based implementations.

The language goes beyond the permission/prohibition concepts of security and shows how to express and implement the obligation concept using ACID transactions.

We have defined an easily described language, which does not have any specific predicate for any type of policy, yet it is able to express RBAC, history-based, obligation-based and delegation policies, among others. The language is composed of rules to decide about accesses, that can be composed using a simple logic with four basic operators (`AND`, `OR`, `NOT` and restriction) and their respective quantifiers (`FORALL`, `EXIST` and restriction quantifier), and uses policies to provide encapsulation, inheritance and parameterization, thus improving policy reusability.

Preliminary results show that SPL should be mainly used to assemble big policy blocks. Simple rules and quantifiers should only be added if absolutely necessary, because they tend to clutter the policy with small and difficult things to read. We are currently developing a graphical interface which is expected to solve some of the low level usability problems identified in SPL [25].

With this language, we have addressed the problems of expressiveness, design and enforcement scalability of access control policies. However, we have not addressed the problem of establishing trust, which is a fundamental precondition for the application of access control policies in large distributed environments. We

have plans to build a trust framework to work together with the access control framework already defined. We also plan to apply SPL to a medium size organization (~ 5000 users) to identify larger library policies thus simplifying future policies design.

Acknowledgements. The authors should like to express their gratitude to Pedro Gama for his value contribute to the development of SPL, to Patricia Lima for the careful review of the paper and to the anonymous referees, who have made this a stronger and more precise paper.

References

[1] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, November 2000.

[2] S. Farrell *at al.* AAA authorization requirements. RFC2906, 2000.

[3] Steven M. Bellovin. Distributed firewalls. *;login: magazine, special issue on security*, November 1999.

[4] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, September 1998.

[5] Elisa Bertino, Francesco Buccafurri, Elena Ferrari, and Pasquale Rullo. A logical framework for reasoning on data access control policies. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 175–191. IEEE Computer Society Press, July 1999.

[6] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A system to specify and manage multipolicy access control models. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 116–127, Monterey, CA, USA, Jun 2002. IEEE Computer Society Press.

[7] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, February 2003.

[8] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 94–109, Oakland, CA, May 1996. IEEE Computer Society Press.

[9] William R. Bevier and William D. Young. A constraint language for Adage. Technical report, Computational Logic, Inc., April 1997.

[10] M. Blaze, J. Feigenbaum, and J. Ionnidis. The KeyNote trust-management system version. Technical report, Internet RFC 2704, September 1999.

[11] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996. IEEE Computer Society Press.

[12] Piero Bonatti, Sabrina de Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, February 2002.

[13] David F. Brewer and Michael J. Nash. The Chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989. IEEE Computer Society Press.

[14] B. Chellas. *Modal Logic*. Camdbridge University Press, 1980.

[15] Fang Chen and Ravi Sandhu. Constraints for role-based access control. In *ACM Workshop on Role-Based Access Control*, 1995.

[16] Jason Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT-03)*, pages 43–50, New York, June 2–3 2003. ACM Press.

[17] Frederic Cuppens and Claire Saurel. Specifying a security policy: A case study. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 123–135. IEEE Computer Society Press, 1996.

[18] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Policy 2001: Workshop on Policies for Distributed Systems and Networks*, pages 17–28, Bristol, UK, January 2001. Springer-Verlag LNCS 1995.

[19] Pedro Dias, Carlos Ribeiro, and Paulo Ferreira. Enforcing history-based security policies in mobile agent systems. In *Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 231–234, Lake Como (Italy), 2003. IEEE Computer Society Press.

[20] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *5th ACM Conference on Computer and Communications Security*, pages 38–48, San Francisco, California, November 1998. ACM Press.

[21] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. Internet RFC 2693, September 1999.

[22] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[23] Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.

[24] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. Heimdhal: A history-based policy engine for grids. In *6th IEEE International Symposium on Cluster Computing and Grid*, Singapore, May 2006. IEEE.

[25] Pedro Gama, Carlos Ribeiro, and Paulo Ferreira. A scalable history-based policy engine. In *IEEE Workshop on Policies for Distributed Systems and Networks*, Ontario CANADA, 2006. IEEE. (to apear).

[26] Simon Godik and Tim Moses. OASIS eXtensible access control markup language (XACML). OASIS draft version 16, August 2002.

[27] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[28] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–14. IEEE Computer Society Press, May 2000.

[29] IBM Corporation. *Entrepise Security Architecture Using Tivoli Security Solutions*, 2002.

[30] Sotiris Ioannidis. *Security Policy Consistency and Distributed Evalution in Heterogeneous Environments*. PhD thesis, University of Pennsylvania, 2005.

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

[32] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–43, Oakland, CA, May 1997. IEEE Computer Society Press.

[33] H. James, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. Technical Report CSE-98-3, University of California, Davis Department of Computer Science, 1998.

[34] A. D. Keromytis, S. Ioannidis, M. Greenwald, and J. Smith. The STRONGMAN architecture. In *Proceedings, DARPA Information Survivability Confernce and Exhibition*, volume 1, pages 178–188. IEEE press, April 2003.

[35] K. Loney and G. Koch. *Oracle 8i – The Complete Reference*. McGraw Hill, 2000.

[36] Emil Lupu and Morris Sloman. A policy based role object model. In *First International Enterprise Distributed Object Computing Workshop (EDOC'97),*, pages 36–47, Gold Coast, Queensland, Australia, October 1997.

[37] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[38] Carlos Ribeiro. *Uma Plataforma Para Políticas de Autorização Para Organizações*. PhD thesis, Instituto Superior Técnico, Lisbon, Portugal, July 2002.

[39] Carlos Ribeiro, Paulo Ferreira, André Zúquete, and Paulo Guedes. Security policy consistency. In *CL2000 - First Workshop on Rule-Based Constraint Reasoning and Programming*, Imperial College, London, UK, July 2000.

[40] Carlos Ribeiro, André Zúquete, and Paulo Ferreira. Enforcing obligation with security monitors. In *The Third International Conference on Information and Communication Security (ICICS'2001)*, pages 172–176, Xi'an, China, November 2001. Springer Verlag.

[41] Carlos Ribeiro, André Zúquete, Paulo Ferreira, and Paulo Guedes. Spl: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS'01)*, San Diego, California, February 2001.

[42] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[43] R. Sandhu. Separation of duties in computarized information systems. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, pages 179–190, Halifax, UK, September18–21 1990.

[44] R. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[45] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[46] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[47] R. Simon and M. E. Zurko. Adage: An architecture for distributed authorization. Technical report, OSF Research Institute, Cambridge, 1997.

[48] R. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 183–194. IEEE Computer Society Press, June 1997.

[49] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[50] Duminda Wijesekera and Sushil Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and System Security*, 6(2):286–325, 2003.

[51] T. Y. C. Woo and S. S. Lam. Authorization in distributed systems: A formal approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 33–51. IEEE Computer Society Press, May 1992.

[52] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chug. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security*, 6(3):404–441, 2003.

[53] Mary Ellen Zurko, Rich Simon, and Tom Sanfilipo. A user-centered, modular authorization service built on an RBAC foundation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 57–73. IEEE Computer Society Press, May 1999.