A Formal Language for Specifying Policy Combining Algorithms in Access Control

Ninghui Li[1] Qihua Wang[1] Prathima Rao[1] Dan Lin[1] Elisa Bertino[1] Jorge Lobo[2]

Abstract

Many access control policy languages, e.g., XACML, allow a policy to contain multiple sub-policies, and the result of the policy on a request is determined by combining the results of the sub-policies according to some policy combining algorithms (PCAs). Existing access control policy languages, however, do not provide a formal language for specifying PCAs. As a result, it is difficult to extend them with new PCAs. The lacking of a formal approach also makes the design of combining algorithms in XACML plagued with issues and subtleties that can be confusing and surprising for policy authors. Motivated by the need to provide a flexible and user-friendly mechanism for specifying PCAs, we propose a policy combining language PCL, which can succinctly and precisely express a variety of PCAs. We show that our approach avoids the pitfalls of XACML and that it is expressive enough to express both PCAs in XACML and other natural PCAs. A policy evaluation engine only needs to understand PCL to evaluate any PCA specified in it. In particular, we model the evaluation of PCAs using finite state automata. Using techniques from automata theory, we also develop systematic policy evaluation optimization techniques that improve evaluation efficiency.

1 Introduction

Many access control policy languages allow a policy to contain multiple sub-policies, and the effect of the policy on a request is determined by combining the effects of the sub-policies according to some strategy. Examples of such policy languages include XACML [17], XACL [8], EPAL [1], SPL [13], and firewall policies. Existing languages usually specify a fixed set of policy combining algorithms (PCAs), but none of them provides a formal language for specifying new PCAs. For example, firewall policy languages typically use the first-applicable strategy to combine results from rules; that is, the decision for a request is decided by the first rule applicable to the request. Among existing policy languages, XACML offers the most flexible approach which consists of four strategies: deny-overrides, permit-overrides, first-applicable, and only-one-applicable. However, many other interesting strategies exist as shown by the following examples.

Weak consensus. Sub-policies should not conflict with each other: Permit a request if some sub-policies permit a request, and no sub-policy denies it. Deny a request if some sub-policies deny

[1]CERIAS and Department of Computer Science, Purdue University, West Lafayette, IN
[2]IBM Watson Research Center, Hawthorne, NY

a request, and no sub-policy permits it. Yield a value indicating conflict if some permit and some deny.

Strong consensus. All sub-policies must agree: Permit a request if all sub-policies permit a request. Deny a request if all sub-policies deny a request. Yield conflict otherwise.

This differs from weak-consensus because a sub-policy may neither permit nor deny a request, i.e., it may not be applicable to the request. When some sub-policies permit a request and others are not applicable to it, weak consensus permits the request, but strong consensus yields conflict.

Weak majority. A decision (permit or deny) wins if it has more votes than the opposite. Permit (deny, resp.) a request if the number of sub-policies permitting (denying, resp.) the request is greater than the number of sub-policies denying (permitting, resp.).

Strong permit majority. Require majority to permit a request. Permit a request if over half sub-policies permit it, and deny the request otherwise.

Furthermore, for each of the four strategies XACML supports, there are several variants that differ in some details, as we will show later in this paper. For each strategy, XACML chooses one variant while other variants appear equally (and sometimes more) appropriate. Therefore, it is important that such variants can also be specified and used.

XACML explicitly allows additional user-defined combining algorithms. However, it does not provide a standard approach (or specification language) for doing so. In fact, even the standard XACML PCAs are specified using natural language and pseudo-code, which are not suitable for automatic processing. Note that for any PCA to be effective, it must be understood by the policy evaluation engine. Without a formal specification language, each new PCA must be hard-coded by programmers in every policy evaluation engine. This makes wide deployment of user-defined PCAs infeasible in practice.

Motivated by the need to provide a flexible and user-friendly mechanism for specifying PCAs, we propose the Policy Combining Language (PCL), which can succinctly and precisely express a variety of PCAs. We show that all the standard combining algorithms in XACML as well as variants of them can be specified in PCL. Also, many other algorithms like those based on weak consensus and strong consensus can be specified by PCL. By using PCL, the policy evaluation engine now only needs to understand one language and can easily evaluate policies using any PCA specified in this language. In particular, the evaluation of PCAs specified in PCL is modeled as a finite state automata. Based on automata theory, we develop novel evaluation techniques that can optimize the evaluation of any PCA according to its formal specification. The link we established between PCL and the theory of formal languages via automata theory enables us to study the expressive power and limitation of PCL. We show that strategies that require counting, such as weak majority and strong permit majority, cannot be expressed in PCL. We then discuss possible extensions to PCL to express such strategies and the limitations of these extensions. Our contributions are summarized as follows:

- We provide a detailed analysis of policy combining algorithms in XACML, identify several problematic cases, and pinpoint their causes.
- We develop a simple and flexible language PCL. The core of a PCA specified in PCL is a binary policy combining operator (PCO) that combines two policies. A binary PCO is extended in a natural way to combine multiple policies. We use a novel approach of systematically treating policy evaluation errors as uncertainty over a set of possible values. Though PCL is primarily motivated by XACML, the underlying formalism and approaches can be applied to other languages.
- We develop a generic evaluation optimization technique. It can automatically optimize the evaluation of any PCA specified in PCL. For the standard PCAs in XACML, our approach performs significantly more kinds of optimizations than current XACML implementations [15, 7]. In other words, existing XACML implementations can benefit from our techniques even before adopting PCL.
- We also propose an XML syntax for PCL and ways for PCL to be integrated to XACML.

The rest of this paper is organized as follows. In Section 2, we analyze PCAs in XACML. In Section 3, we introduce the language PCL. We then discuss optimization techniques for policy evaluation in Section 4. In Section 5, we discuss the expressive power of PCL and extensions to PCL. In Section 6, we discuss how this work may impact XACML. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Policy Combination in XACML

In this section, we analyze policy combining strategies adopted in XACML. Our descriptions are based on XACML 2.0 [17], and they remain accurate for the current draft XACML 3.0. We first illustrate the complexity and subtleties of policy combining in XACML, analyze its design rationale, and then identify several critical issues and pinpoint the causes of those issues. Such analysis helps illustrate the challenges and requirements that need to be addressed by PCL.

2.1 XACML Policy Structure

XACML [17] is the OASIS standard language for the specification of access control policies.

Rules, policies, and policy sets. XACML defines three policy elements: *rule*, *policy*, and *policy set*. A rule is the most basic policy element; it has three main components: a *target*, a *condition*, and an *effect*. The target defines a set of subjects, resources and actions that the rule applies to; the condition specifies restrictions on the attributes in the target and refines the applicability of the rule; the effect is either `Permit`, in which case we call the rule a *permit rule*, or `Deny`, in which case we call it a *deny rule*. If a request satisfies both the rule target and rule condition, the rule is applicable to the request and yields the decision specified by the effect element; otherwise, the rule is not applicable to the request and yields the decision `NotApplicable`.

A policy consists of four main components: a *target*, a *rule-combining algorithm (RCA)*, a set of *rules*, and *obligations*. The policy target decides whether a request is applicable to the policy and it has similar structure as the rule target. The RCA specifies how the decisions from the rules are combined to yield one decision. The *obligations* represent functions to be executed in conjunction with the enforcement of an authorization decision. We ignore *obligations* in this paper as it is not related to policy combining in XACML.

A policy set also has four main components: a *target*, a *policy-combining algorithm (PCA)*, a set of *sub-policies*, and *obligations*. A sub-policy can be either a policy or a policy set. The PCA specifies how the results of evaluating the sub-policies are combined to yield one decision.

Dataflow model. A dataflow model for XACML is given in Figure 1. (This shows only elements relevant to our discussion. The complete dataflow model [17] contains more elements.) Major actors in the dataflow model are the Context Handler, the Policy Enforcement Point (`PEP`), the Policy Decision Point (`PDP`), and the Policy Information Point (`PIP`).

When an access request is received, the `PEP` sends the request to the `PDP` via the context handler. To determine whether a request should be permitted, the `PDP` may need additional information about subjects, resources, actions and environment attributes from the `PIP`. Once the `PDP` receives the required information from the `PIP`, it makes a decision according to the policies and returns the result to the `PEP`. Depending on the response received from the `PDP`, the `PEP` may either permit the request or deny it.

Several kinds of errors may occur during policy evaluation. Some are due to network communication and database querying. For example, the `PIP` may be down and thus unable to answer queries from the `PDP`. Some are due to erroneous policies. For example, the condition of a rule may perform a division by 0.

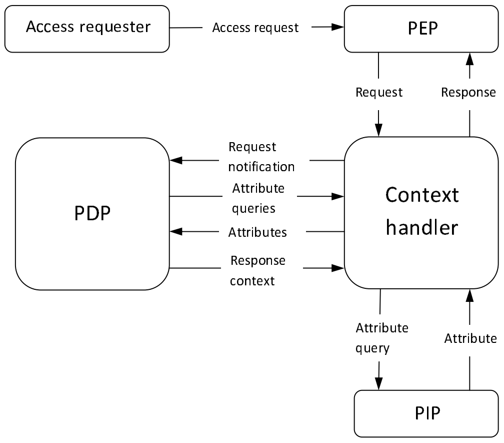


Figure 1: XACML dataflow diagram

Others are due to missing attributes. For example, evaluating a policy that denies a request if the subject's income level is below a threshold yields an error if the income information is not provided. When this policy is a sub-policy of a policy-set that permits a request if the subject is a member of a particular club, the income information is not necessary for the overall policy to permit the request. Hence, a requestor may provide only proof of his club membership but no information about his income, resulting in an evaluation error. In all the above cases, the `PDP` does not have enough information to determine whether a policy applies to the access request. It is the responsibility of the PCA to handle such errors.

2.2 Rule and Policy Combining Algorithms

In XACML, a rule, a policy, or a policy set returns one of the following four decisions for each request: P (`Permit`), D (`Deny`), NA (`NotApplicable`), and Ind (`Indeterminate`). The value Ind is overloaded with several meanings. It can represent an error, or an unresolved conflict during policy evaluation.

For one request, there may be multiple rules and policies that are applicable, and they may yield different decisions. To resolve the conflict of those decisions, XACML provides five standard RCAs and six standard PCAs. They are *"Deny-overrides"*, *"Ordered-deny-overrides"*, *"Permit-overrides"*, *"Ordered-permit-overrides"*, *"First-applicable"* and *"Only-one-applicable"* ("Only-one-applicable' is only defined as a PCA). Ordered-deny-overrides and ordered-permit-overrides are the same as deny-overrides and permit-overrides, respectively, except that rules and policies have to be evaluated in the order they appear.

While the intuition of these combining algorithms are easy to understand and formalize, the full specifications of these algorithms are quite complicated and have subtleties which are often ignored by existing attempts to formalize them. In what follows we illustrate the subtleties and problematic cases of these algorithms, and show that these are caused by the overloading of Ind.

Deny-overrides RCA. The "Deny-overrides" RCA prefers D to P to NA. That is, if any rule evaluates to D, the result is D; otherwise if no rule evaluates to D, some rules evaluate to P, and the rest evaluate to NA, the result is P; and if all rules evaluate to NA, the result is NA. We write this as $\mathsf{D} > \mathsf{P} > \mathsf{NA}$.

When the evaluation of a rule encounters an error, the RCA treats the result of this rule as Ind. The treatment of Ind, however, depends on whether the Ind is from a permit rule or a deny rule. When the Ind is from a permit rule, the RCA uses $\mathsf{D} > \mathsf{P} > \mathsf{Ind} > \mathsf{NA}$, and when the Ind is from a deny rule, the RCA uses

$\mathsf{D} > \mathsf{Ind} > \mathsf{P} > \mathsf{NA}$.

The reason for this treatment is not explained in the official XACML standard specifications. Our understanding is as follows. When a permit rule results in Ind, this means that the evaluation of the target or condition of the rule encounters an error; as a result, we are uncertain whether this rule is applicable or not. When this rule is applicable, this rule returns P; when this rule is not applicable, this rule returns NA. These are the only two cases, and in either case, we can conclude that the request could be permitted if another permit rule applies to the request. In other words, we can view Ind here as representing uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$, and because we have $\mathsf{P} > \mathsf{NA}$, we can derive $\mathsf{P} > \mathsf{Ind} > \mathsf{NA}$. When Ind is from a deny rule, it is treated as uncertainty over $\{\mathsf{D}, \mathsf{NA}\}$. Since it is possible that the request will be denied by this deny rule if there is no error, P cannot override Ind, but D can; hence $\mathsf{D} > \mathsf{Ind} > \mathsf{P}$.

Issue 1 *In XACML,* Ind *is treated differently depending on which context* Ind *is from. This complicates the specification and understanding of the RCAs.*

The above issue is caused by the fact that Ind is overloaded. At the level of rules, Ind can represent either uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$ or uncertainty over $\{\mathsf{D}, \mathsf{NA}\}$.

Deny-overrides PCA. Like its RCA counterpart, "Deny-overrides" PCA also uses $\mathsf{D} > \mathsf{P} > \mathsf{NA}$; however, it treats Ind differently. The PCA treats Ind as always equivalent to D. Presumably, this is because of the following reasons. The PCA combines results from multiple policies, and an Ind value from a policy could be due to various reasons like errors or conflicts. As the PCA does not have enough knowledge about the exact cause of the Ind, it conservatively treats this as a potential deny. Further, since deny-overrides prefers deny, this PCA treats a potential deny as deny.

However, treating Ind as D is problematic. A more natural order could be $\mathsf{D} > \mathsf{Ind} > \mathsf{P}$ which is consistent with the previous RCA case (when Ind is caused by error in a deny rule). This difference becomes significant when we consider the interaction between the `PDP` and the `PEP` in Section 2.3.

To illustrate why XACML's design is undesirable, we consider the following example, where a request is denied by XACML, but arguably should be permitted.

Example 1 *Consider a request q and the following policy set:*

$$S = \textit{Deny-overrides}(P_1 = \textit{Deny-overrides}(R_1, R_2), \quad P_2),$$

where S consists of two policies P_1 and P_2, and P_1 consists of two rules R_1 and R_2 (see Figure 2). Suppose that R_1 is a deny rule that does not apply to q, R_2 is a permit rule the evaluation of which on q encounters an error, and P_2 is a policy that permits q. Thus, R_1 gives NA, *and R_2 gives* Ind*; according to RCA semantics, P_1 gives* Ind. *According to the PCA semantics, S denies the request. However, this is undesirable for the following reasons. Among R_1, R_2, and P_2, the only deny rule (R_1) does not apply to q and P_2 permits q. There is no reason for denying the request. This becomes clearer if we consider the* cause *of the value* Ind *which is the uncertainty of whether R_2 is applicable or not. When R_2 is applicable, R_2 permits the request, so does P_1. Since both P_1 and P_2 permit the request, so should S. When R_2 is not applicable, the policy P_1 is not applicable. Since P_2 permits the request and* $\mathsf{P} > \mathsf{NA}$ *in deny-overrides, S should permit the request.*

While whether the request should be permitted or denied in the above example may be subject to debate, the fact that one can plausibly argue for permit while XACML gives deny indicates problems, because a policy author may expect one result while getting the other. We summarize this problem as Issue 2.

Issue 2 *There are plausible ways to treat* Ind *that are not allowed by XACML.*

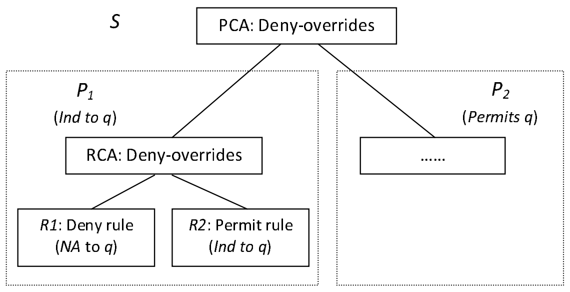


Figure 2: Graphical representation of policy set S in Example 1.

We observe that this problem is again caused by the overloading of Ind. At the rule level, one can find out the underlying causes of Ind, and handle it accordingly. At the policy level, one does not know what Ind represents, and thus has to treat it with insufficient knowledge.

Permit-overrides. The "Permit-overrides" RCA is similar to the "Deny-overrides" RCA, except that the preference is $\mathsf{P} > \mathsf{D} > \mathsf{NA}$. The Ind is also treated similarly as uncertainty. That is, if Ind is from a permit rule, we have $\mathsf{P} > \mathsf{Ind} > \mathsf{D} > \mathsf{NA}$, and if Ind is from a deny rule, we have $\mathsf{P} > \mathsf{D} > \mathsf{Ind} > \mathsf{NA}$.

The "Permit-overrides" PCA uses $\mathsf{P} > \mathsf{D} > \mathsf{Ind} > \mathsf{NA}$ to handle Ind. Here, we have an asymmetry between "Deny-overrides" PCA and "Permit-overrides" PCA. The former treats Ind as D (and is preferred over P), while the latter not only doesn't treat Ind as P but also treats it as less preferred than D. Recall that the cause of an Ind could be uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$, preferring D over Ind goes against the name "Permit-overrides". One could argue that $\mathsf{P} > \mathsf{Ind} > \mathsf{D} > \mathsf{NA}$ is a more natural choice for the "Permit-overrides" PCA. In any case, this asymmetry may be confusing and surprising for policy authors.

Issue 3 *In XACML, the treatments of* Ind *in Permit-overrides and Deny-overrides are asymmetric, while the two algorithms give the impression of being symmetric.*

First-applicable. The "First-applicable" RCA returns the effect of the first applicable rule as the combining result if no errors occur. Whenever an error occurs during the rule evaluation, the RCA returns Ind. One could easily argue that the "First-applicable" RCA should be defined in a different way. Following the spirit that an error occurring at a permit rule evaluation means uncertainty about $\{\mathsf{P}, \mathsf{NA}\}$, if the next rule permits the request (it has a permit effect and is applicable), the policy should return P (rather than Ind), as the request will be permitted whether the first rule applies or not. This is another case of Issue 2. The same argument applies to the "First-applicable" PCA.

Only-one-applicable. The "Only-one-applicable" PCA returns the effect of the unique policy in the policy set which applies to the request. If there are more than one applicable policies, the PCA reports the conflict by returning Ind. Furthermore, if an error occurs during evaluation of any policy, the PCA also returns Ind. We can observe that the meanings of Ind in the above two cases are different. The former represents a conflict that the PCA cannot resolve, and the latter reflects an uncertainty caused by an error.

Empty policies. When a policy or a policy set is empty, the combining result is always NA, according to the pseudo-code of the combining algorithms in XACML.

2.3 Interaction Between PDP and PEP

The `PDP` returns a value in $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{Ind}\}$ to the `PEP`, and the `PEP` decides whether to permit or deny a request. XACML defines three types of `PEP`s: *Base `PEP`* , *Deny-based `PEP`* , and *Permit-based `PEP`* . All `PEP`s yield the permit (or deny) decision if the value returned by the `PDP` is P (or D). The difference among these `PEP`s lies in the way they handle NA and Ind. The Base `PEP`'s behavior is undefined when receiving NA or Ind. The Deny-based `PEP` treats NA and Ind as `Deny`, while the Permit-based `PEP` treats NA and Ind as `Permit`.

Consider again Example 1, for which we argued that the request should be permitted whereas XACML returns D. If one used the Permit-based `PEP` and changed the definition of the "Deny-overrides" PCA to $\mathsf{D} > \mathsf{Ind} > \mathsf{P} > \mathsf{NA}$, then the request would be permitted. However, there are still other situations in which a request should arguably be denied but is permitted by the `PEP`.

The problem is due to the fact that, as we have seen earlier, the returning result Ind has very different meanings for different PCAs or even for the same PCA. For example, Ind may represent an error on policy target evaluation, an error when evaluating a `Deny` rule (which implies a potential `Deny`), an error when evaluating a `Permit` rule (which implies a potential `Permit`), a conflict when using "Only-one-applicable" PCA (more than one policy are applicable, could be two `Deny`, two `Permit`, or one `Deny` and one `Permit`). It is unclear why the Deny-based `PEP` (or the Permit-based `PEP`) treats all the cases in the same way.

Issue 4 *There are plausible interactions between `PDP` and `PEP` that cannot be achieved in XACML, because when the Ind value is returned by `PDP`, the `PEP` has too little information.*

3 The Policy Combining Language PCL

Solving the aforementioned problems in XACML and being able to express a wider class of PCAs are the main design goals of PCL. More specifically, we have the following design requirements.

1. One should be able to specify the exact behavior of current XACML PCAs and RCAs, even though we have argued that some behavior may be less desirable than other alternatives. This is necessary for backward compatibility, which we believe is critical for ease of deployment and adoption of PCL in XACML.
2. One should be able to specify some of the alternatives that we argue to be more natural (e.g., $\mathsf{P} > \mathsf{Ind} > \mathsf{D} > \mathsf{NA}$ for permit-overrides PCA).
3. One should be able to specify PCAs that consider the root cause of Ind and resolve it accordingly. For example, the deny-overrides-based PCA should permit the request as shown in Example 1.
4. One should be able to specify as many other natural PCAs as possible, such as the ones presented in Section 1.
5. The specification should be concise, precise, and user-friendly.

Requirement 5 conflicts with the other 4 requirements on expressive power. PCL achieves requirements 1, 2, and 3, and can express weak consensus and strong consensus as presented in Section 1. However, in order to keep PCL simple and user-friendly, we sacrifice some expressive power, and PCL is thus unable to express weak majority or strong permit majority, which requires the ability to count. We will discuss how to extend PCL to express the two strategies in Section 5.

In the rest of this section, we present details of PCL. Unlike XACML, PCL does not distinguish the specification of RCA and PCA; they are specified in the same way in PCL. Therefore from now on, we use the term PCA to mean both RCA and PCA.

3.1 Overview of PCL

As discussed in Section 2, the overloading of Ind causes many problems during policy combining in XACML. A straightforward approach is to replace Ind with different values in different situations. For example, one can use a value to denote conflict when two policies are applicable to the request and the only-one-applicable PCA is adopted, another value to denote uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$, and yet another value to denote uncertainty over $\{\mathsf{D}, \mathsf{NA}\}$. Using such a six-valued approach, however, has a number of problems.

First, the more values are used, the more difficult it is to specify and understand a PCA. Second, treating uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$ as a new value, say P/NA, means that one could specify how P/NA combines with other values in an arbitrary way. For example, suppose that the combination result of NA and D and the combination result of P and D are both defined as D. Considering the inherent semantic requirement, the combination of $\{\mathsf{P}, \mathsf{NA}\}$ with D should be D. However, treating P/NA as a new value, one can specify the combination result to be any value other than D. Third, six values are insufficient for representing all situations during policy evaluation. Consider the case of combining the value of uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$ with the value of uncertainty over $\{\mathsf{D}, \mathsf{NA}\}$ using the deny-overrides PCA. The combination result should be the uncertainty over $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}\}$ when performing the case by case analysis and we can observe that this is a new situation not represented by any of the six values. Similarly, when using the first-applicable PCA, an uncertainty over $\{\mathsf{P}, \mathsf{NA}\}$ followed by a D represents an uncertainty over $\{\mathsf{P}, \mathsf{D}\}$, which is yet another new situation. Note that the value $\{\mathsf{P}, \mathsf{D}\}$ here does not mean a conflict between P and D. It indicates that it is uncertain whether P or D should be the final result, but it is certain that only one of them is the result.

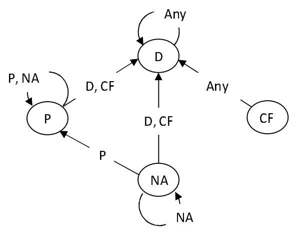
Our solution to this problem is to use only four values $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$, and to represent evaluation errors as *uncertainty* denoted by non-singleton subsets of Σ. Here the value CF means conflict. It represents a conflict between two policies (or rules) that a PCA chooses not to resolve but to expose to a higher level. Some example situations where conflicts may occur are listed as follows: (1) when the only-one-applicable PCA is used, there will be a conflict if more than one policies apply to the request; (2) when the weak-consensus PCA is used, there will be a conflict if one policy permits while another denies; (3) when the strong-consensus PCA is used, there will be a conflict if one policy permits while another does not apply. The reason we introduce CF is because Ind is overloaded and using CF is conceptually clearer. To integrate PCL into XACML with minimal changes to XACML, one could use Ind to encode CF. We discuss this in Appendix A.

Next, we present the technical foundations of PCL.

Policy combining operators. First, consider the case when a PCA is used to combine two policies (or rules) assuming error-free evaluation. We call such a PCA a policy combining operator (PCO). A PCO g can be represented as a binary operator g over the four values $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$, i.e., $g : \Sigma \times \Sigma \rightarrow \Sigma$. A binary operator over the four values $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$ can be expressed using a matrix. Some examples of the matrix-representation of PCOs are shown in Figure 3.

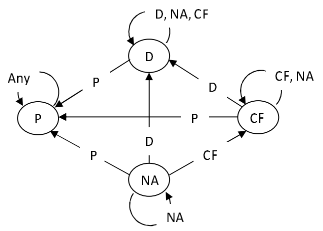
While a matrix has 16 cells, in practice one does not have to give 16 pieces of information to specify one PCO. More compact representations are possible. One can specify some matrices using an ordering, e.g., $\mathsf{P} > \mathsf{D} > \mathsf{CF} > \mathsf{NA}$ for permit overrides. Or, one can exploit the fact that some matrices are symmetric. One can also exploit the fact that all cells in some row or column of a matrix have the same value. Our proposed

P1 \ P2	P	D	NA	CF
P	P	D	P	D
D	D	D	D	D
NA	P	D	NA	D
CF	D	D	D	D



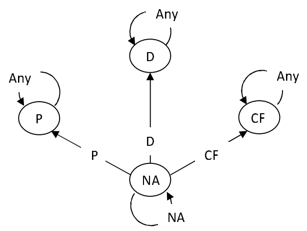
(a) Deny-overrides

P1 \ P2	P	D	NA	CF
P	P	P	P	P
D	P	D	D	D
NA	P	D	NA	CF
CF	P	D	CF	CF



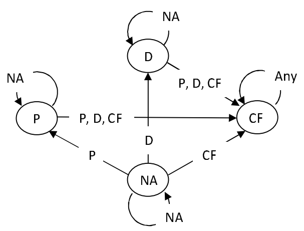
(b) Permit-overrides

P1 \ P2	P	D	NA	CF
P	P	P	P	P
D	D	D	D	D
NA	P	D	NA	CF
CF	CF	CF	CF	CF



(c) First-applicable

P1 \ P2	P	D	NA	CF
P	CF	CF	P	CF
D	CF	CF	D	CF
NA	P	D	NA	CF
CF	CF	CF	CF	CF



(d) Only-one-applicable

Figure 3: Figure of several PCOs in matrix and the corresponding general PCAs in the form of DFA.

XML encoding of PCL, which will be discussed in Appendix A, uses these observations in specifying PCOs.

From PCO to PCA. In general, the number of policies a PCA needs to combine cannot be bounded at the time when the PCA is specified. That is, a PCA should be a function $\Sigma^+ \to \Sigma$, where

$$\Sigma^+ = \Sigma \bigcup \Sigma \times \Sigma \bigcup \Sigma \times \Sigma \times \Sigma \bigcup \cdots$$

Therefore, we propose the following approach to construct a general PCA from a PCO.

Definition 1 *Let* $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. *Given a PCO* $g : \Sigma \times \Sigma \to \Sigma$, *its* recursive *PCA is the function* $f : \Sigma^+ \to \Sigma$ *defined as follows:*

- $f(P_1) = P_1$
- $f(P_1, P_2) = g(P_1, P_2)$
- $f(P_1, \ldots, P_n) = g(f(P_1, \ldots, P_{n-1}), P_n)$, *for* $n > 2$.

According to this definition, combining a sequence of policies proceeds from the first to the last. The first two are first combined, the result of which is combined with the third, and so on. An alternative is define $f(P_1, \ldots, P_n) = g(P_1, f(P_2, \ldots, P_n))$. This expresses the same PCAs as Definition 1 when the operator g is associative. All PCOs that we have encountered are associative. We choose our approach because the first to last ordering is more commonly used for combing policies. Also, when an operator is both commutative and associative, one can combine the sub-policies in any order.

One can view the evaluation of a policy with such a PCA against a request q as a deterministic finite automaton (DFA). The current state is a value in $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$, representing the result of combination so far, and the input symbol is also one value in Σ, representing the result of the next rule or policy on q. When the evaluation terminates, the state of the automaton is the result of the combination. Examples of the DFA-representation of four standard PCAs in XACML are shown in Figure 3. We will discuss how empty policies (policies that contain no sub-policies) are handled in Section 3.2.

Note that the DFA corresponding to a PCA has Σ as its set of input symbols, and $\Sigma \cup \{\mathsf{S}\}$ as its set of states, where S denotes the start state. The transitions out of S is fixed for all PCAs; on symbol $x \in \Sigma$, it will go to the state x. Therefore, we omit the start state in Figure 3. All states in the DFA are accept states; however, on which accept state a DFA stops is significant. While a standard DFA defines one language, our automaton simultaneously defines four languages, each corresponding to a state in $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. For instance, if a string in Σ^+ leads the automaton to end at state P (meaning the result of combination is to permit the request), then that string is in the language corresponding to P.

Handling evaluation errors using uncertainty. So far we have discussed the PCA under the assumption that there are no evaluation errors. We now take into account such errors. In particular, we treat errors as uncertainty and allow each rule/policy/policy-set to evaluate to a subset of $\{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. If an error occurs when evaluating a permit rule, the result will be $\{\mathsf{P}, \mathsf{NA}\}$ which means it is uncertain whether this permit rule permits or is not applicable to the request. Similarly, a deny rule evaluates to $\{\mathsf{D}, \mathsf{NA}\}$ when evaluation of its target or condition encounters an error. The following definition specifies how to combine results that represent uncertainty.

Definition 2 *Let* $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. *Let* Γ *denote the set containing all non-empty subsets of* Σ, *i.e.,* $\Gamma = 2^\Sigma \setminus \emptyset$. *The PCO* g *is extended to be a binary function* $g : 2^\Sigma \times \Gamma \to \Gamma$, *defined as follows.*

$$\begin{aligned} g(\gamma_1, \gamma_2) &= \{g(\sigma_1, \sigma_2) \mid \sigma_1 \in \gamma_1 \wedge \sigma_2 \in \gamma_2\}, && \text{for } \gamma_1, \gamma_2 \in \Gamma \\ g(\emptyset, \gamma) &= \gamma, && \text{for } \gamma \in \Gamma \end{aligned}$$

For example, assume that g is a “Deny-overrides” PCO (its matrix representation is shown in Figure 3). We have

$$\begin{aligned} g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{D}\}) &= \{g(\mathsf{P}, \mathsf{D}), g(\mathsf{NA}, \mathsf{D})\} &&= \{\mathsf{D}\} \\ g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{P}\}) &= \{g(\mathsf{P}, \mathsf{P}), g(\mathsf{NA}, \mathsf{P})\} &&= \{\mathsf{P}\} \\ g(\{\mathsf{P}, \mathsf{NA}\}, \{\mathsf{NA}\}) &= \{g(\mathsf{P}, \mathsf{NA}), g(\mathsf{NA}, \mathsf{NA})\} &&= \{\mathsf{P}, \mathsf{NA}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{D}\}) &= \{g(\mathsf{D}, \mathsf{D}), g(\mathsf{NA}, \mathsf{D})\} &&= \{\mathsf{D}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{P}\}) &= \{g(\mathsf{D}, \mathsf{P}), g(\mathsf{NA}, \mathsf{P})\} &&= \{\mathsf{D}, \mathsf{P}\} \\ g(\{\mathsf{D}, \mathsf{NA}\}, \{\mathsf{NA}\}) &= \{g(\mathsf{D}, \mathsf{NA}), g(\mathsf{NA}, \mathsf{NA})\} &&= \{\mathsf{D}, \mathsf{NA}\} \end{aligned}$$

It is worth noting that the behavior of the deny-overrides and permit-overrides RCA in XACML (as we discussed in Section 2.2) can be explained using the above approach, assuming that uncertainty is treated as Ind.

Given a PCO g, the behavior of the PCA using g can still be modeled using the finite state automata for g. However, an important difference is that now each input to be combined may be a set consisting of two or more values, e.g., $\{\mathsf{P}, \mathsf{NA}\}$. As a result, the next state may also be uncertain. We call this an uncertain-input FSA. The behavior of such a automaton can be equivalently simulated by a DFA with 16 states and 15 input symbols, where each state is a subset of $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$ and each input is a non-empty subset of Σ. The initial state is the empty set $\emptyset$, and the state transitions can be defined according to Definition 2. Given a state γ_1 and the input γ_2, the new state is given by $g(\gamma_1, \gamma_2)$. All 16 states are accept states. Each non-empty state defines a different language over Σ^+.

We observe that an uncertain-input FSA is conceptually related to but different from non-deterministic finite state automaton (NFA), where the state transition is non-deterministic. In an uncertain-input FSA, the state transition relation is deterministic, but we may have uncertain inputs. Converting an NFA to DFA increases the number of states, but not the number of input symbols. Converting an uncertain-input FSA to a DFA increases both the number of states and the number of input symbols.

We stress that our approach is different from an approach using 15 values. Our approach requires a 4-by-4 matrix for combining the four basic values, and infer the combining behavior of other, uncertain values. Compared with a 15-valued approach, our approach implicitly rules out many 15-by-15 matrices as illegal, and represent the legal ones compactly.

3.2 Additional Details of PCL

We have introduced the foundations of PCL. While our design of PCL is motivated by XACML, it can be used in any policy language that needs to combine a possibly unbounded number of policies and/or needs to consider policy evaluation errors. Next, we will discuss some additional issues that arise mostly for compatibility considerations with XACML.

Order-preserving evaluation. XACML has ordered deny-overrides and permit-overrides. To be compatible, PCL provides an option to specify whether the evaluation must follow the order of the original rules (policies).

Pre-processing. We use uncertainty to handle errors occurring at the time of evaluation. However, to specify PCAs to behave exactly the same as those in XACML (assuming that Ind is used to encode CF), we need to allow errors to be handled in ways other than uncertainty. For example, using the first-applicable RCA in XACML, if we have two permit rules such that the first one results in an error and the second one is applicable, then our approach returns P. In order to return Ind (CF, actually) in this case, the information that an error occurred needs to be maintained. To do so, we introduce an optional pre-processing step to a PCA. One can specify that an uncertainty value resulted from a sub-policy is to be converted to one of P, D, CF or NA before feeding it to the PCO. For instance, to simulate the behavior of first-applicable RCA in XACML using a PCA in PCL, one specifies a pre-processing step that maps all uncertain values to CF.

Note that a PCA with a pre-processing step will not have any uncertain input, and hence we can use the 5-state DFA (the five states are $\{S\} \cup \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$) for policy evaluation.

In addition to achieving backward compatibility with XACML, pre-processing can also be useful for a PCA that does not expect evaluation errors in the sub-policies and wants to always propagate errors up as CF.

Post-processing. While the pre-processing step applies to each input result to be combined, the post-processing step applies to the final combining result.

There are two reasons for introducing a post-processing step in a PCA. The first is to deal with empty policies. In XACML, when there is no result to be combined, all PCAs result in NA. In PCL, an empty policy results in $\emptyset$, which we did not define as a valid input value in Definition 2. $\emptyset$ thus needs to be mapped to another value. When no post-processing specification is given, the default behavior is to map $\emptyset$ to $\{\mathsf{NA}\}$.

The second reason is to enable interactions with existing `PEP`s. If a `PDP` returns an uncertain result to a `PEP` that follows the current XACML standard, the `PEP` is not able to understand the result. In a post-processing specification, one can specify whether/how some of 16 possible results should be mapped to one of P, D, NA, CF. Moreover, one can also specify whether the post-processing mapping should be always applied or only be applied when the result is to be returned to the `PEP`.

We would like to point out that a `PDP` which returns the result without post-processing mapping allows more flexibility on the `PEP`-side. For example, finer-grained permit-based `PEP` can be defined in similar ways as the post-processing specification. One example `PEP` is as follows: Upon a returned decision $\gamma \subseteq \Sigma$, permit the access if and only if γ contains P but not D.

3.3 Expressing PCAs using PCL

In this section, we show that PCL can express the exact behavior of all the standard combining algorithms provided by XACML as well as several other combination strategies. When expressing XACML we assume that `Indeterminate` is used to replace CF.

(a) Deny Overrides RCA

P1\P2	P	D	NA	CF
P	P	D	P	
D	D	D	D	
NA	P	D	NA	
CF				

(b) Permit Overrides RCA

P1\P2	P	D	NA	CF
P	P	P	P	
D	P	D	D	
NA	P	D	NA	
CF				

(c) Weak Consensus PCA

P1\P2	P	D	NA	CF
P	P	CF	P	P
D	CF	D	D	D
NA	P	D	NA	CF
CF	CF	CF	CF	CF

(d) Strong Consensus PCA

P1\P2	P	D	NA	CF
P	P	CF	CF	CF
D	CF	D	CF	CF
NA	CF	CF	NA	CF
CF	CF	CF	CF	CF

(e) Last-Applicable PCA

P1\P2	P	D	NA	CF
P	P	D	P	CF
D	P	D	D	CF
NA	P	D	NA	CF
CF	P	D	CF	CF

Figure 4: PCOs in matrix using PCL

The XACML deny-overrides RCA can be expressed using the matrix in Figure 4(a), with no pre-processing and a post-processing step that maps non-singleton sets to CF. The XACML deny-overrides RCA essentially treats errors as uncertainty (see our discussion in Section 2.2), which is the same as PCL; hence, no pre-processing is needed to map uncertain values to CF. Note that CF can never occur as an input, because it cannot be the result of the evaluation of a rule; hence we do not need to specify the column for CF. Further, as CF does not occur in the first three rows, it can never be reached from the starting state; hence we do not need to specify the row for CF.

The XACML deny-overrides PCA can be expressed using the matrix in Figure 3(a), with a pre-processing step that maps all non-singleton sets (denoting evaluation errors) to CF, and a post-processing step that maps CF to D. The XACML permit-overrides RCA can be expressed using the matrices in Figure 4(b), with no pre-processing and default post-processing (mapping $\emptyset$ to NA). The XACML permit-overrides PCA can be expressed using the matrix in Figure 3(b), with a pre-processing step that maps uncertain values to CF and default post-processing.

Ordered versions of the above RCAs and PCAs in XACML can be expressed by indicating order-preserving evaluation in corresponding PCA specifications.

The first-applicable RCA and PCA can be expressed using the matrix in Figure 3(c), with a pre-processing step that maps non-singleton sets to CF and default post-processing. The only-one-applicable PCA can be expressed using the matrix in Figure 3(d), with the same pre-processing and post-processing as above.

Weak consensus and strong consensus can be expressed using the matrices in Figures 4 (c) and (d) respectively, with a pre-processing step that maps uncertain values to NA for weak consensus and to CF for strong consensus, and both with default post-processing. And a "last-applicable" strategy can be expressed as well, using the matrix in Figure 4(e), even though policy combination occurs from first to last.

4 Optimization Techniques

In many situations, one does not need to combine the results of all sub-policies before being able to reach a decision for a request. For example, in first-applicable PCA, one could return the decision of the first sub-policy that is applicable. This can be viewed as an optimization in the evaluation of the first-applicable PCA. The types of optimizations that can be performed depend on the nature of the PCA and the combining result so far. Several such optimizations are present in the pseudo-code descriptions of the PCAs and RCAs in the XACML standard. They are designed by humans based on analysis of the PCAs. PCL enables the specification of new PCAs; thus a challenging and interesting question is whether we can perform optimizations automatically without human analysis of the PCAs. In this section, we develop a systematic approach for optimized evaluations using techniques from automata theory.

4.1 Optimization Examples

Before presenting our algorithm for optimized evaluations for PCL, we first look at a number of example cases where optimizations can be performed.

Terminating values. In some PCAs, once a certain value is encountered, the result will never change, no matter what values will be encountered in the future. As a result, one can stop the evaluation and return. Examples of such cases include D for deny-overrides; P for permit-overrides; P, D, and CF for first-applicable; and CF for only-one-applicable and strong consensus. Terminating values can be discovered from the PCO matrix. A value $\sigma \in \Sigma$ is terminating if the row corresponding to it contains only σ.

Rule skipping. When a policy evaluation reaches certain state, there is no need to evaluate the remaining permit rules (or deny rules). For example, when weak-consensus is used and the current state is P, i.e., one applicable permit rule has been found, there is no need to evaluate other permit rules, one only needs to evaluate deny rules.

Rule reordering. For some RCAs, it is beneficial to evaluate deny rules before permit rules (or the other way around). For example, when the deny-overrides RCA is used, deny rules should be evaluated before permit rules; because once a deny rule is found to be applicable, the final result is determined, but even after a permit rule is found to be applicable, one still needs to evaluate the deny rules.

Nested evaluations. Nested policies provide additional opportunities for optimization. For example, consider a policy that uses the permit-overrides PCA to combine its sub-policies. Suppose that the first sub-policy denies the request, then the state is $\{\mathsf{D}\}$. For the next sub-policy, the evaluation engine only needs to determine whether it yields a $\{\mathsf{P}\}$, a non-singleton set containing $\{\mathsf{P}\}$, or a subset of $\{\mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. The state will go to $\{\mathsf{P}\}$, $\{\mathsf{P}, \mathsf{D}\}$, or $\{\mathsf{D}\}$ respectively. Suppose that the next sub-policy uses a permit overrides RCA, and one has evaluated all permit rules and has found that none of the permit rules is applicable. Normally, the engine needs to evaluate the deny rules to determine whether to return D, NA, or CF. In this case, however, the engine can skip all the deny rules, because the result will be a subset of $\{\mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$.

For another example, consider a policy that uses the only-one-applicable PCA in XACML, after the engine has seen one sub-policy that permits or denies the request, it only needs to be able to tell whether other sub-policies are applicable or not. This enables a number of optimizations when evaluating these sub-policies. For instance, if a sub-policy uses the permit-overrides RCA, then the engine can stop after any evaluation error, which will result in the overall PCA returning CF. For another instance, if a sub-policy (a policy-set) uses the only-one-applicable PCA, then one can stop after any permit or deny is encountered.

Optimizations in XACML implementations. The pseudo-code in the XACML standard performs the "terminating values" optimization, but not the other optimizations discussed above. The two XACML implementations we have examined [15, 7] follow the pseudo-code in the standard, and do not perform optimizations beyond terminating values.

4.2 Optimization Algorithms

Our optimization algorithm is based on the DFA-representation of a PCA. As we have seen in Section 3.1, the evaluation procedure of a policy can be viewed as state transition in a DFA. When no pre-processing is specified, policy evaluation errors are handled using uncertainty, and evaluating the PCA is modeled by a 16-state DFA. When pre-processing is specified, policy evaluation errors are converted to one of the four values, and evaluating the PCA is modeled by a 5-state DFA. Given an arbitrary PCA specified in PCL, the algorithm constructs its corresponding DFA, and analyzes the properties of the DFA, and then uses these properties to optimize the evaluation. This way, the algorithm can automatically optimize any PCA specified in PCL.

In what follows, we present the main idea of our optimization techniques. Detailed algorithms can be found in Figures 7 and 8 in Appendix B.

DFA Minimization. In a 16-state DFA, often times one does not need to distinguish among all 16 outcomes. When post-processing is used, the 16 outcomes are partitioned into equivalence classes that are mapped to the same value in the post-processing step. For example, in XACML deny-overrides RCAs, all non-singleton outcomes are mapped to $\{\mathsf{CF}\}$. We can exploit this fact to simplify the DFA, reducing the number of states and helping triggering other optimizations that we will discuss later.

We use standard DFA minimization techniques to do this. We first calculate the set of all states reachable from the starting state $\emptyset$. We then partition these reachable states into groups based on the post-processing step. If two states are mapped to the same value in post-processing, then they are in the same group. States in the same group are marked as "indistinguishable", while states in different partitions are marked as "distinguishable". Then, for each pair of indistinguishable states S_1 and S_2, we examine every possible input to see if there is an input that leads the two states into two distinguishable states. If such an input exists, we mark S_1 and S_2 "distinguishable". We then keep repeating the second step until all pairs of states are distinguishable or no more pair of distinguishable states can be found. Finally, we merge indistinguishable states in a DFA.

This DFA minimization step needs to be performed when the PCA is first encountered. One can store the result and reuse it when encountering the PCA again. We point out that when one uses a PCA to combine rules, one knows that there are only five possible outcomes from these rules; they are $\{\mathsf{P}\}$, $\{\mathsf{D}\}$, $\{\mathsf{NA}\}$, $\{\mathsf{P}, \mathsf{NA}\}$, and $\{\mathsf{D}, \mathsf{NA}\}$. This knowledge can help the DFA minimization process, as one only needs to consider these five inputs, rather than all 16 inputs. This may result in more compact DFAs. Given a PCA, one can create two minimized versions of it, one for combining rules, where there are five possible inputs, and one for combining general policies (where there are 16 possible inputs).

In the discussions below, we assume that policy evaluation uses such minimized DFAs.

Terminating values. Given a DFA, a state is a terminating state if there is no edge leaving the state. This indicates a terminating value, and the evaluation can terminate once such a state is reached.

Rule and policy evaluation skipping. When encountering a sub-policy (which can be a rule, a policy, or a policy-set in XACML), one knows that the sub-policy will result in one of the 16 values without evaluating it. Sometimes, however, one is able to narrow this down to a subset of the 16 values without evaluating it. This may enable us to skip the evaluation of the sub-policy. For example, for a permit rule, its result will be $\{\mathsf{P}\}$, $\{\mathsf{NA}\}$, $\{\mathsf{P}, \mathsf{NA}\}$. If all three outcomes result in changing to the same state (e.g., all three result in remaining at the current state), then one can skip the evaluation and perform the state transition directly. For a sub-policy that uses a PCA that has a post-processing step, the post-processing step may limit the possible outcomes of the sub-policy, which may enable skipping the evaluation of this sub-policy. We point out that the rule skipping optimization in Section 4.1 is a special case of this.

Reordering. One can change the ordering of evaluation only when two conditions are satisfied. First, the PCA is not order-preserving. Second, the PCO specified by the matrix is both commutative and associative. A PCO is commutative if and only if its matrix is symmetric. The associativity of a PCO can also be determined based on its matrix.

Rules are clearly classified into permit rules and deny rules, but policies cannot be easily classified. Because of this, the reordering optimization is likely to be useful only when combining rules. Sometimes it may be advantageous to evaluate all permit rules first or all deny rules first. For example, for deny-overrides

RCA, it is better to first evaluate all deny rules. This can be identified from the DFA by recognizing that one kind of rules (e.g., deny rules) can lead to a terminating state (at least one of the possible outcomes leads to a terminating state), whereas another kind of rules cannot lead to a terminating state (no possible outcome leads to a terminating state).

Input partitions. Finally, we discuss techniques to enable optimizations of nested evaluations. The key observation is as follows: When the DFA of the parent PCA is at a given state, certain inputs are equivalent in that they will lead the DFA to the same resulting state. Hence one can partition the 16 outcomes into equivalence classes. When evaluating the next sub-policy, one only needs to tell which equivalence class the result falls in, rather than the precise result. For example, if a parent PCA is only-one-applicable, and the current state is either $\{\mathsf{P}\}$ or $\{\mathsf{D}\}$, the 16 outcomes are divided into two equivalence classes: $\{\mathsf{NA}\}$ and everything else. The former makes the state stay unchanged, while the latter results in $\{\mathsf{CF}\}$, a terminating value. When evaluating a sub-policy, one needs only to tell in which class the result falls.

Our algorithm utilizes this information by passing the equivalence classes into the function that evaluates the sub-policy. This information is then used to further minimize the DFA for the sub-policy, which has already been minimized based on post-processing. If in the DFA, two states fall into the same equivalent class, then it may be possible to merge them. For example, if one only needs to differentiate between $\{\mathsf{NA}\}$ and everything else, the DFA for permit-overrides PCA needs only two states, one for $\{\mathsf{NA}\}$ and the other for the remaining cases, and the second state is a terminating state. This further minimization enables more optimizations based on terminating values and rule and policy evaluation skipping.

This DFA minimization step is performed based on partitions determined by where the PCA appears, thus it cannot be performed using only the description of the PCA. However, the minimization of such a small DFA (between 4 and 16 states) takes little time. Also, one can cache the minimization result of a DFA using a particular partition using a hashtable, and reuse the result when the same situation is encountered.

5 PCL's Expressive Limitations and Extensions

A natural question that arises is: How expressive is PCL? This question has both practical and theoretical relevance. In this section, we first analyze the limitations of PCL and then introduce an extension to PCL, which can express weak majority and strong permit majority strategies.

5.1 PCL's Expressive Limitations

For clarity, we assume that policy evaluation errors are always handled by uncertainty and ignore policy evaluation errors in the discussions below. To understand what PCAs can be expressed in PCL and what cannot, we need a definition of PCAs independent of PCL. This is given below.

Definition 3 *A PCA is given by a tuple of four languages $\langle L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{CF}}\rangle$ over the alphabet of $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$, satisfying the conditions that they are mutually disjoint and $\Sigma^{+} = L_{\mathsf{P}} \cup L_{\mathsf{D}} \cup L_{\mathsf{NA}} \cup L_{\mathsf{CF}}$.*

For example, the first-applicable PCA can be expressed using regular expression syntax as:

$$\begin{array}{lll} \langle & L_{\mathsf{P}} = \mathsf{NA}^{*}\,\mathsf{P}\,\Sigma^{*}, & L_{\mathsf{D}} = \mathsf{NA}^{*}\,\mathsf{D}\,\Sigma^{*}, \\ & L_{\mathsf{NA}} = \mathsf{NA}^{*}, & L_{\mathsf{CF}} = \mathsf{NA}^{*}\,\mathsf{CF}\,\Sigma^{*} \quad \rangle. \end{array}$$

Viewing PCAs as languages gives us access to the tools developed in formal languages and automata theory.

Definition 4 *We say a PCA $\langle L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{CF}}\rangle$ is regular if and only if the languages $L_{\mathsf{P}}, L_{\mathsf{D}}, L_{\mathsf{NA}}, L_{\mathsf{CF}}$ are regular.*

The following (obvious) proposition establishes an upper-bound on the kind of PCAs that can be expressed in PCL.

Proposition 1 *All PCAs that can be specified in* PCL *are regular.*

Proof. All PCAs specified in PCL are expressed using DFA, and the languages derived from DFAs are regular. ∎

This proposition immediately rules out certain PCAs to be expressible in PCL, because they are not regular.

Corollary 2 *Weak majority PCA and strong permit majority PCA cannot be expressed in* PCL .

Proof. Weak majority and strong permit majority are not regular, because they require counting to an unbounded number. A more formal proof is omitted here. It can be shown using the pumping lemma for regular languages. ∎

While the set of regular PCAs is an upperbound of what can be expressed in PCL, the bound is not tight. Not all regular PCAs can be expressed, because PCAs in PCL use only five states.

Proposition 3 *There exist regular PCAs that cannot be expressed in PCL.*

For example, consider the following policy combining strategy.

At least k. Permit a request if at least k sub-policies permit it. Deny a request if at least k deny. Return CF if both of the above two conditions hold, and return NA if neither holds.

This is regular for any fixed k, as one can use a DFA which counts how many P's and D's have been encountered so far. We point out that "at least 1" is the same as weak-consensus, which is expressible in PCL. However, one cannot express "at least 2" in PCL, since it requires differentiating between eight states:

$$\{\langle i \times \mathsf{P}, j \times \mathsf{D}\rangle \mid i, j \in \{0, 1, 2+\}\},$$

whereas PCL can distinguish only four of them. Only the four states in Σ can be used to differentiate states. If one wants to use the initial state to encode $\langle 0\times\mathsf{P}, 0\times\mathsf{D}\rangle$, then because it must go to NA when encountering NA, NA must also denote $\langle 0 \times \mathsf{P}, 0 \times \mathsf{D}\rangle$.

Finally, we observe that if a PCA can be expressed using a four-state DFA, such that each of the four states is the accept state for one of the four languages, then one can convert the DFA into a 4×4 matrix and express it in PCL.

5.2 Extensions to PCL

Weak majority and strong permit majority appear quite natural, especially weak majority. We now explore how to extend PCL to express them. Both strategies require counting. The standard way to do counting in an automaton is to add a stack, such as a pushdown automaton does. One approach is to keep the four states we have in PCL and introduce an additional stack. The downside of this approach is that the behavior is determined by the current state, the current top of stack, and the current input symbol. Specifying such behavior requires a 4-by-4-by-4 cube, which we feel may be too clumsy to specify and difficult to understand.

We thus take the approach of not having a current state and using only the stack to maintain the current evaluation progress. A next question is: When the evaluation stops, what value should be returned? We take the approach of returning the top of the stack. An alternative approach is to define a separate PCO for processing the content of the stack at the end of evaluation and returns an answer. This is more complicated, but may offer more expressive power. As it is unclear whether such expressive power is needed, we did not choose this approach.

Intuitively, to implement weak majority or strong permit majority, when encountering a P with an empty stack, we will push P onto it; if we then encounter D, we need to pop P so that they cancel out. Here we have a difficulty. The result should be CF because we have an equal and non-zero number of P's and D's. Yet the stack is empty, the same situation as an empty policy, for which we want to return NA. To deal with this, we introduce a new symbol B that always appear at the bottom of a stack after the evaluation has encountered at least one input.

The extension, which we call PCL^+, still uses a matrix to specify a PCO g; however, there are two differences from a matrix in PCL. First, a matrix for a PCO contains an additional row for B. Second, cells in the matrix can contain two extra symbols: push and pop, in addition to $\Sigma = \{\mathsf{P}, \mathsf{D}, \mathsf{NA}, \mathsf{CF}\}$. That is, a PCO in PCL^+ is given by a function:

$$g : \Sigma \cup \{\mathsf{B}\} \times \Sigma \to \Sigma \cup \{\mathsf{push}, \mathsf{pop}\}.$$

Policy evaluation in PCL^+ uses a stack. Given the next value $y \in \Sigma$ to be combined, there are five cases.

- *Start:* If the stack is empty, then push first the value B and then the value y onto the stack.
- *Bottom:* If the top of the stack is B, then push $g(\mathsf{B}, y)$ onto the stack.
- *Push:* Otherwise, let x be the value at the top of the stack. If $g(x, y) = \mathsf{push}$, push y onto the stack.
- *Pop:* If $g(x, y) = \mathsf{pop}$, pop x out from the stack.
- *Replace:* If $g(x, y) \in \Sigma$, replace the current top (i.e., x) with $g(x, y)$.

PCL can be viewed as a special case of PCL^+, where the matrix does not contain push or pop; as a result, after the "Start" step, only replacing occurs, and B is never at the top of the stack.

To deal with policy evaluation errors, the evaluation state needs to contain a set of stacks, rather than a single stack. When one encounters a policy error interpreted as uncertainty over a set of values, one duplicates the current evaluation state (i.e., the stacks) for each additional value in the set, and evaluates the duplicated stacks for that value. After processing each input, multiple stacks with the same contents are merged.

Both weak majority and strong permit majority can be specified in PCL^+. The weak majority PCA can be specified using the matrix in Figure 5 (a), with no pre-processing and a post-processing step that maps B to CF. The strong permit majority PCA can be specified using the matrix in Figure 5 (b), with no pre-processing and a post-processing step that maps everything other than P to D.

(a) Weak Majority

P1\P2	P	D	NA	CF
P	push	pop	P	P
D	pop	push	D	D
NA	P	D	NA	CF
CF	P	D	CF	CF
B	P	D	CF	CF

(b) Strong Permit Majority

P1/P2	P	D	NA	CF
P	push	pop	pop	pop
D	pop	push	push	push
NA	pop	push	push	push
CF	pop	push	push	push
B	P	D	NA	CF

Figure 5: PCOs in matrix using PCL^+

There are PCAs that cannot be expressed using PCL^+. Consider a "Strong majority" PCA, which returns P if more than half of the policies return P, returns D if more than half of the policies return D, and returns NA otherwise. Intuitively, in order to determine whether more than half of the policies return

P, we make use of the stack to match P with everything else; and in order to determine whether more than half of the policies return D, we use the stack to match D with everything else. However, with only one stack in PCL$^+$, we cannot perform matching for both P and D, and at the very beginning of the combining procedure there is no way to determine whether we should match CF and NA with P or D.

The optimized evaluation techniques we have developed in Section 4 need to be revised for PCL$^+$. This is a technically challenging and interesting topic and is left for future work.

6 Impacting XACML

While PCL provides a general approach for specifying PCAs, and is useful for any access control policy language that desires sophisticated policy combining behavior, the place that PCL is mostly likely to have an immediate impact is XACML. We have developed an XML encoding of PCL, which is described in Appendix A. XACML has been increasingly adopted in recent years, and is becoming the de facto standard for specifying access control policies for various applications, especially web services. Currently, version 3.0 of XACML is being developed. We plan to submit our proposal of adding PCL to the XACML standard group.

Before PCL is adopted by XACML, this paper can impact practice in three ways. First, the discussions of XACML PCAs and RCAs in this paper can help improve the understanding of them. Second, XACML PDP implementations can use the techniques in this paper to optimize the evaluation of XACML policies. Third, some new PCA strategies, such as weak-consensus and strong consensus, can be adopted by future XACML standards as standard PCAs and RCAs.

A basic core of PCL can be introduced to XACML by allowing PCAs and RCAs specifications in PCL using `Indeterminate` to encode CF. PDP implementations using PCL need to be updated; however PEP implementations remain unchanged. Hence the same PEP can be served by updated and current versions of PDPs.

A more thorough level of adoption introduces the value CF, allows the interactions between `PDP` and `PEP` to use sets of values for uncertainty, and allows more flexible types of `PEP`s than the current three standard types: basic `PEP`, deny-based `PEP`, and permit-based `PEP`. New types can be defined in ways similar to the post-processing step of PCAs in PCL.

Whether adopting PCL$^+$ would prove to be useful seems unclear at this moment. Writing PCAs and RCAs with a stack-based evaluation model seems to be too cumbersome and prone to specification mistakes. We need to investigate if there are simpler abstractions that expand the expressiveness of PCL.

7 Related Work

Many policy languages specify some fixed policy combining behaviors, such as XACML [17], XACL [8], EPAL [1], SPL [13], and firewall policies. Policy combing also appears in the access control language in Bauer et al. [3]. However, none of them provides a formal language for specifying new PCAs. The problem of combining access control policies has raised significant interests in the research community [5, 6, 2, 9, 10, 12, 16]. Our work is more practical and comprehensive. We deal with three important issues that are not considered in previous approaches. First, we are able to combine an *unbounded* number of sub-policies. Second, we deal with policy evaluation errors in a principled way. Third, we have a comprehensive approach to optimized evaluation.

Bonatti et al. [5] proposed a 2-valued algebra for composing access control policies. Backes et al. [2] introduced a 3-valued algebra for combining enterprise privacy policies. Their algebra applies to EPAL [1] policies, whose effect can be “allow”, “deny”, or “don’t care”. They defined operators such as conjunction, disjunction, and scoping in the algebra. Bruns el at [6] introduced a 4-valued algebra for policy combination. They defined a policy as a 4-valued predicate that maps each request to grant, deny, conflict, or unspecified, which correspond to the four elements of the Belnap bilattice [4]. In these approaches, policy composition are expressed using an algebraic term. Since we allow a matrix to be used to specify PCOs, one could express any algebraic term in PCL, though this may involve a nested policy with different PCAs. For example, a term $P_1 + (P_1 - P_3)$, where $+$ and $-$ are two operators, can be expressed using $f_1(P_1, f_2(P_1, P_3))$, where f_1 and f_2 are two PCAs using PCOs corresponding $+$ and $-$. Using algebraic terms, however, one cannot express how to combine an unbounded number of sub-policies. These approaches also do not deal with evaluation errors.

Mazzoleni et al. [12] proposed a policy integration algorithm for XACML. Their approach allows an resource owner to specify how she would like her policies to be combined with third party policies, and the combination behavior can be based on analyzing the two policies to be combined. For example, one can specify that two policies are combined using some strategy only if they are similar according to some measurement of similarity. Such behavior cannot be specified in PCL, as the combining result on one request may be depending on how the policies handle other requests. This work, however, proposes only new combination algorithms, rather than a language for specifying such algorithms.

Kolovski et al. [10] presented a formalization of XACML using description logics. They map the semantics of XACML policies and combining algorithms to a set of logical rules, which allows them to perform a variety of analysis on XACML policies. The formalization of XACML combining algorithms in this work, however, is incomplete. It does not deal with policy evaluation errors, nor can it express the only-one-applicable PCA. The formalization in [10] was not intended to be a language for specifying PCAs and cannot deal with unbounded number of sub-policies. Wijesekera and Jajodia [16] proposed a propositional algebra to manipulate access control policies. An interesting aspect about this work is that a policy can be a non-deterministic set of permission assignments, e.g., a policy may grant either permission A or permission B but not both. They also use algebraic approach for policy combining, and hence has the same limitations as other algebraic approaches.

Our work uses automata for policy evaluation. This is related to work on security automata [14, 11], started by Schneider [14]. Inputs to a security automaton are actions on the objects guarded by the corresponding policy. If the automaton can make a transition on an input, then the target is allowed to perform that step; otherwise, the security policy is violated. A security automaton in [14] describes the sequence of actions that are allowed by one policy. We use automaton to combine results from sub-policies, where each input is a decision rather than an action. We are using similar techniques (automata, in particular) to solve different problems in access control.

8 Conclusion

In this paper, we have introduced an expressive policy combining language PCL. PCL can combine an *unbounded* number of sub-policies, deals with policy evaluation errors in a principled approach, and has a comprehensive approach to evaluation optimization. One big advantage of having such a formal language is that it enables the introduction and usage of new PCAs. Anyone can create a new PCA, and all PCL-enabled policy evaluation engines will be able to evaluate policies using the PCA. Moreover, PCL overcomes many problems occurring in XACML and our evaluation optimization techniques based on automata theory enable new classes of optimizations. Our novel approaches of using uncertainty to handle evaluation errors and using DFAs to model policy combination can be applied in any access control policy language that needs sophisticated policy combining behavior. In particular, we are working on impacting the XACML standard.

Acknowledgments The work reported in this paper has been partially supported by: IBM under the OCR project Privacy and Security Policy Management; the National Science Foundation under Grant No.0430274. We thank Ian Molloy and Yuqing Sun for their useful suggestions on the improvement of the paper.

References

[1] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. The enterprise privacy authorization language (EPAL). *http://www.w3.org/2003/p3p-ws/pp/ibm3.html*.

[2] M. Backes, M. Durmuth, and R. Steinwandt. An algebra for composing enterprise privacy policies. In *ESORICS '04: Proceedings of the 2004 European Symposium on Research in Computer Security*, 2004.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: ACM Conference on Programming Language Design and Implementation*, 2005.

[4] N. D. Belnap. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, 1977.

[5] P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):1–35, Feb. 2002.

[6] G. Bruns, D. S. Dantas, and M. Huth. A simple and expressive semantic framework for policy composition in access control. In *FMSE '07: Proceedings of the 2007 ACM Workshop on Formal methods in security engineering*, pages 12–21, New York, NY, USA, 2007. ACM.

[7] Google Code. Enterprise Java XACML implementation. *http://code.google.com/p/enterprise-java-xacml/*.

[8] S. Hada and M. Kudo. XML access control language: Provisional authorization for XML documents. *http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html*.

[9] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *CSFW '03: Proceedings of the Computer Security Foundations Workshop*, 2003.

[10] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 677–686, New York, NY, USA, 2007. ACM.

[11] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. 2003.

[12] P. Mazzoleni, E. Bertino, B. Crispo, and S. Sivasubramanian. XACML policy integration algorithms: not to be confused with XACML policy combination algorithms! In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 219–227, New York, NY, USA, 2006. ACM.

[13] C. Ribeiro, A. Zquete, P. Ferreira, and P. Guedes. SPL: An acess control language for security policies with complex constraints. In *NDSS '01: Network and Distributed System Security Symposium*, 2001.

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

[15] Sun Microsystems. Sun's XACML implementation. *http://sunxacml.sourceforge.net/*.

[16] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and Systems Security (TISSEC)*, 6(2):286–325, May 2003.

[17] XACML TC. OASIS eXtensible Access Control Markup Language (XACML). *http://www.oasis-open.org/committees/xacml/*.

A XML Encoding for PCL

In this section, we present an XML encoding for PCL so that it can be easily integrated into existing XACML implementations. A PCA specification is composed of an optional *Pre-processing* element, a *DecisionMatrix* or *DecisionOrder* element and an optional *Post-processing* element, which has the following format:

```
PCA :=
   <PolicyCombiningAlgorithm
     policyCombiningAlgId=name
     preserveOrder=(yes|no)>
       (pre-processing)?
       (decision-matrix | decision-order)
       (post-processing)?
   </PolicyCombiningAlgorithm>
```

policyCombiningAlgID denotes the name of the policy combining algorithm and *preserveOrder* denotes whether rules/policies need to be evaluated in their listed order.

The *Pre-processing* element is encoded as follows:

```
pre-processing :=
   <Pre-processing>
     (<Map-To-Permit/> | <Map-To-Deny/> |
     <Map-To-NotApplicable/> | <Map-To-Conflict/>)
   <Pre-processing/>
```

A PCO can be specified either as a matrix using the *DecisionMatrix* element or as some operations on a given order of preference over the possible decisions using the *DecisionOrder* element. The encoding of the *DecisionMatrix* is as follows.

```
decision-matrix :=
   <DecisionMatrix isSymmetric=(yes|no)>
     <Permit-Result>
       (decision-pair)+
```

        <Permit-Result/>
        <Deny-Result>
            (**decision-pair**)+
        <Deny-Result/>
        <NotApplicable-Result>
            (**decision-pair**)+
        <NotApplicable-Result/>
        <Conflict-Result>
            (**decision-pair**)+
        <Conflict-Result/>
    <DecisionMatrix/>

decision-pair := <tag_1−tag_1/>

tag_1 := (Permit|Deny|NotApplicable|Conflict|Any)

Note that the *Permit-Result*, *Deny-Result*, *NotApplicable-Result* and *Conflict-Result* elements denote the possible values in each cell of the matrix and they may or may not be present. The **decision-pair** elements denote the specific cells of the matrix. The attribute *isSymmetric* indicates whether the matrix is symmetric or not. For a symmetric matrix, we only need to specify cells in the half triangle which saves some efforts when writing a new PCA.

The *DecisionOrder* element is encoded as follows:

decision-order :=
    <DecisionOrder
        operator=(`Maximum`|`Minimum`)>
        <Preference> < tag_2/ > <Preference/>
        <Preference> < tag_2/ > <Preference/>
        <Preference> < tag_2/ > <Preference/>
        <Preference> < tag_2/ > <Preference/>
    <DecisionOrder/>

tag_2 := (Permit|Deny|NotApplicable|Conflict)

Finally, the optional *Post-processing* element is encoded as follows :

post-processing :=
    <Post-processing>
        <Permit-Result>
            (**one-of**)* | (**any-of**)?
        <Permit-Result/>
        <Deny-Result>
            (**one-of**)* | (**any-of**)?
        <Deny-Result/>
        <NotApplicable-Result>
            (**one-of**)* | (**any-of**)?
        <NotApplicable-Result/>
        <Conflict-Result>
            (**one-of**)* | (**any-of**)?

```
<PolicyCombiningAlgorithm
        policyCombiningAlgId="deny-overrides"
        preserveOrder="no"/>
    <Pre-processing> <Map-To-Deny/>
    <Pre-processing/>
    <DecisionMatrix isSymmetric="yes">
        <Permit-Result>
            <Permit-Permit/> <Permit-NotApplicable/>
        <Permit-Result/>
        <Deny-Result>
            <Deny-Any/> <Conflict-Any/>
        <Deny-Result/>
        <NotApplicable-Result>
            <NotApplicable-NotApplicable/>
        <NotApplicable-Result/>
        <Conflict-Result> <Conflict-Result/>
    <DecisionMatrix/>
<PolicyCombiningAlgorithm/>
```

Figure 6: XML specification of the Deny-Overrides PCA

<Conflict-Result/>
<Indeterminate-Result>
(**one-of**)* | (**any-of**)?
<Indeterminate-Result/>
<Post-processing/>
one-of := <OneOf> $< tag_2 / >$ <OneOf/>
any-of := <AnyOf> ($< tag_2 / >$)+ <AnyOf/>

Note that the inclusion of each of the result sub-elements is optional. The behaviour with respect to $\emptyset$ can be defined by having a empty result element. The *OneOf* and *AnyOf* elements offer flexibility in terms of specifying the set of decisions to be mapped. The *OneOf* elements can be used to specify a particular set of decisions to be mapped while the *AnyOf* element can be used to specify some decisions that must appear in the the set of decisions to be mapped.

To exemplify, figure 6 shows the standard deny-overrides PCA using the proposed XML syntax. The ordered-deny-overrides can be similarly specified with the preserveOrder attribute set to the value "yes". Figure 4 shows decision matrices of two other example PCAs that can be specified in PCL. These matrices can be easily converted into XML format using the proposed XML. For brevity, we omit their XML presentation here.

B Optimization Algorithms

The algorithms for optimized evaluation are given in Figures 7 and 8.

```
eval_pca(PCA, Partitions, P[])
begin
  //create a DFA to model the PCA and then minimize it
  DFA = new DFA(PCA);
  DFA.minimize(Partitions);

  //evaluate the policies or policy sets
  for(i = 1; i <= P.length; i++)
    //determine if the DFA is in a terminating group
    if (DFA.inTerminiatingGroup())
       return PCA.postMap(DFA.currentState);

    //group inputs that lead the DFA to the same next state
    inputPartitions = dfa.getInputPartitions();

    if (P[i] is policy_set)
      Result = eval_pca(P[i].pca, inputPartitions, P[i].policies);
    else   // P[i] is a policy
      Result = eval_rca(P[i].rca, inputPartitions, P[i].rules);

    //transit to the next state based on the input
    DFA.transit(PCA.preMap(Result));
  endFor;

  return PCA.postMap(DFA.currentState);
end
```

Figure 7: Description of the optimization algorithm for PCA

```
eval_rca(RCA, Partitions, R[])
begin
  //create a DFA to model the PCA and then minimize it
  DFA = new DFA(PCA);
  DFA.minimize(Partitions);

  //evaluate the rules in the policy
  for(i = 1; i < R.length; i++)
       //determine if the DFA is in a terminating group
       if (DFA.inTerminiatingGroup())
           return PCA.postMap(DFA.currentState);

       //perform rule preference and skipping optimization
       //if ordering can be changed
       if (canChangeOrder && RCA.PCO.isCommutativeAndAssociative())
           if DFA.canSkipPermitRules()
               Remove the remaining permit rules;
           if DFA.canSkipDenyRules()
               Remove the remaining deny rules;
           if DFA.preferPermitRules()
               Select the next permit rule and switch it with the current R[i+1];
           if DFA.preferPermitRules()
               Select the next deny rule and switch it with the current R[i+1];
      endIf;
      Result = eval_rule(R[i]);
      DFA.transit(RCA.preMap(result));
  endFor;
  return RCA.postMap(DFA.currentState);
end
```

Figure 8: Description of the optimization algorithm for RCA