Distributed Policy Specification and Interpretation with Classified Advertisements

Nicholas Coleman

West Virginia University Institute of Technology
405 Fayette Pike, Montgomery, WV 25136
Nicholas.Coleman@mail.wvu.edu

Abstract. In a distributed system, the principle of separation of policy and mechanism provides the flexibility to revise policies without altering mechanisms and vice versa. This separation can be achieved by devising a language for specifying policy and an engine for interpreting policy. In the Condor [14] high throughput distributed system the ClassAd language [16] is used to specify resource selection policies and matchmaking algorithms are used to interpret that policy by matching jobs with available machines. We extend this framework to specify and interpret authorization policies using the SPKI/SDSI [6] public key infrastructure. SPKI/SDSI certificates are represented using the ClassAd language and certificate chain discovery is implemented using a modified matchmaking algorithm. This extension complements the resource selection policy capabilities of Condor with the authorization policy capabilities of SPKI/SDSI. Techniques for policy analysis in the context of resource selection and authorization are also presented.

1 Introduction

One of the challenges of distributed computing environments is the specification and interpretation of policy. The separation of policy and mechanism has long been one of the key principles in systems design. This principle simplifies the specification of policies and keeps them independent of implementation changes. One way of achieving separation is to provide a policy framework consisting of a language for specifying policies and an engine for interpreting these policies in the context of a given set of system conditions. The flexibility of such a framework is particularly suitable for resource allocation policy in a distributed system.

Distributed systems are dynamic in that principals and resources may join or leave the federation at any time. Allocation of resources in a decentralized environment requires policy for resource selection and access control. Resource selection is the process of finding resources that satisfy a principal's requests. Access control policies determine whether the principal is permitted to access the resources. Currently there is no single language or framework that deals with authorization and resource selection policies.

C. Russo and N.-F. Zhou (Eds.): PADL 2012, LNCS 7149, pp. 198–211, 2012.

The ClassAd language is based on the concept of classified advertisements. Entities in Condor are represented by classified advertisements or ClassAds. Each job submitted by a condor user has a corresponding ClassAd as does each compute machine. The matchmaking process pairs jobs with machines based on the policies expressed in their ClassAds. Since the bilateral matchmaking framework is not sufficient for assembling three or more parties a multilateral matchmaking framework, *gangmatching*, is required in such cases. A collection of three or more ClassAds that satisfy each others `Requirements` expressions is called a *gang*.

SPKI/SDSI is an infrastructure for expressing authorization policy using public key encryption. Two kinds of certificates can be issued by a principal. An *authorization certificate* grants another principal a set of access rights for a resource as well as the permission to delegate these rights to other principals. A *name certificate* creates a name for another principal or set of principals. A combination of several certificates that authorize a principal to access a resource is called a *certificate chain*. The problem of assembling a suitable *certificate chain* for a given authorization is called the *certificate chain discovery problem* [2].

SPKI/SDSI certificates may be represented using the ClassAd language in a gangmatching context. A ClassAd representing a certificate is composed of several nested ClassAds called *ports*. One of these ports offers the certificate for use in a chain. If needed, additional ports request other certificates to resolve a SPKI/SDSI name or delegate an authorization. A gang of such ClassAds corresponds to a chain of certificates. In order to support the capability to reuse a certificate indefinitely in a chain while avoiding infinite loops, a modified algorithm for gangmatching is presented.

In the case of multilateral matching, two matchmaking analysis problems are presented along with their solutions: *Break the Chain* and *Missing Link*. The *Break the chain* problem occurs when an authorization policy grants an access that needs to be revoked. To revoke an authorization, a set of certificates must be invalidated such that no chain can be constructed granting the authorization. A new algorithm using the results of the gangmatching algorithm identifies a set of ClassAds representing such certificates. The *Missing Link* problem occurs when a desired authorization is not granted by any certificate chain. A modified version of the gangmatching algorithm identifies the additional certificate ClassAds needed to complete a gang representing a chain of certificates granting the desired authorization.

Section 2 describes the ClassAd language and the gangmatching paradigm. Section 3 provides an introduction to the SPKI/SDSI trust management system and describes a ClassAd language representation of SPKI/SDSI certificates. Section 4 discusses the structures and concepts necessary for extending the gangmatching algorithm. The algorithm itself is presented in Sect. 5. Techniques for gangmatching analysis are explored in Sect. 6. Section 7 surveys related work and Sect. 8 concludes the paper.

2 The ClassAd Language and Gangmatching

The ClassAd language is used by Condor primarily to advertise resources and requests for those resources in a distributed environment. An advertisement, called a *ClassAd*, represents an offer of or request for a resource and consists of named descriptive attributes, constraints and preferences. The constraints are expressed by an attribute named `Requirements`, and the expression of the preferences is named `Rank`. [1] A matchmaking process is used to discover offers and requests that satisfy one another's constraints and best suit one another's preferences. If more than two parties are involved – such as a job, a machine, and a license – a bilateral matchmaking scheme is insufficient and a multilateral framework, called *gangmatching* [15], must be used.

In the gangmatching framework a multilateral match is broken down into several bilateral matches. A set of ClassAds that satisfy one another's constraints is called a *gang*. Each ClassAd contains a list of nested ClassAds called *ports*, each of which represents a single bilateral match. A gang is *complete* if all ports of all ClassAds in the gang have been successfully matched to ports of other ClassAds in the gang. A port that has not been matched is an *open* port. Given a port P of a ClassAd and a potentially matching port P' of another ClassAd, a reference in P to an attribute `attr` defined in P' is represented as `other.attr` to distinguish it from a reference to an attribute in P. In addition, P has a label that is used by subsequent ports in the same ClassAd to reference attributes defined in P'. If P's label is `label`, a reference in a subsequent port to an attribute `attr` defined in P' is represented as `label.attr`. The attribute `attr` is *imported* from P' and is called an *imported attribute*.

Figure 1 shows a gangmatching ClassAd representing a job. The ClassAd has two ports: the first requests a machine to run the job, and the second requests a license to run a particular application on that machine. In the `Requirements` expression of the first port of the job ClassAd, a reference to the attribute `Memory`, imported from a matching ClassAd representing a machine, is expressed as `other.Memory`. The port is labeled `cpu`, and the subsequent port contains a reference to the `Name` attribute imported from the ClassAd matching the first port expressed as `cpu.Name`. In contrast, a locally defined attribute like `ImageSize` is referenced locally without using a prefix.

A gang is tree-structured, which means that some ClassAds may not express constraints on other ClassAds directly. For example, in Fig. 1 the job ClassAd contains a port requesting a machine and another port requesting a license. The license and machine ClassAds that match may each contain a port expressing constraints on the job, but may not have ports expressing constraints on one another. This restriction can be circumvented if the job exports attributes imported from the machine ClassAd in the license port. In Fig. 1 the `Name` attribute of the `cpu` ad is exposed in the license port by the definition `CPUName = cpu.Name`. The matching license ClassAd can indirectly reference the `Name` attribute of

[1] To simplify matters this paper deals only with `Requirements` expressions and omits `Rank` expressions from example ClassAds.

```
[Ports = {
  [ // request a workstation
    other = cpu; Type = "cpu_request"; ImageSize = 28M;
    Requirements = other.Type == "Machine" && other.Arch == "INTEL" &&
      other.OpSys == "LINUX" && other.Memory >= ImageSize
  ],
  [ // request a license
    other = license; Type = "license_request"; CPUName = cpu.Name;
    Cmd = "run_sim";
    Requirements = other.Type == "License" && other.App == Cmd
  ]}
]
```

Fig. 1. A gangmatching ClassAd for a job

the machine ClassAd as `other.CPUName`. Circular dependencies are avoided by the restriction that a port may only use imported attributes from previous ports.

3 SPKI/SDSI

SPKI/SDSI is a trust management system that specifies access control policies using certificates. A SPKI/SDSI certificate is a declaration by a principal, the *issuer* of the certificate, about the naming of another principal, the *subject* of the certificate, or the authorization for the subject to access a resource.

Principals are represented by a unique public key. They may also be referred to indirectly by a *SPKI/SDSI name*. A SPKI/SDSI name consists of a public key followed by zero or more identifiers. The identifiers navigate a hierarchical name space, similar to a hierarchical directory structure. For example, if K_A represents the principal named Alice, then the SPKI/SDSI name "K_A Bob Carol" can be resolved by looking up the identifier "Bob" in Alice's namespace. Assuming that K_A Bob resolves to K_B, Bob's public key, the identifier "Carol" must now be looked up in Bob's namespace. If Bob has defined the identifier "Carol" to resolve to K_C, Carol's public key, then "K_A Bob Carol" is equivalent to the SPKI/SDSI names "K_B Carol" and "K_C."

A name certificate (*name cert*) defines a name in the issuer's local name space by assigning an identifier to a SPKI/SDSI name that represents the subject of the certificate. An authorization certificate (*auth cert*) indicates that the issuer (represented by a public key) authorizes the subject (represented by a SPKI/SDSI name) to access a resource. Both the resource and the permission being granted are specified in an auth cert. For the purposes of this paper we are only concerned with a single anonymous resource and a generic operation on that resource. An auth cert also indicates whether or not the authorization may be delegated. In the discussion that follows, we shall adopt the representation of certificates as rewrite

rules with the issuer on the left and the subject on the right as introduced in [2]. Four examples of this rewrite rule representation are shown in Fig. 2.

There are four principals involved in the example certificates in Fig. 2: the administrator of resource R (identified by the public key K_R), Alice, Bob, and Carol (identified by their public keys K_A, K_B, and K_C). Certs (2) and (4) are name certs that indicate that the identifier "Bob" in Alice's name space represents Bob's key, and the identifier "Carol" in Bob's name space represents Carol's key. Certs (1) and (3) are auth certs, denoted by the □ after the subject. In cert (1), the subject "K_A Bob" is granted access to the resource R. The □ at the end indicates that the subject may delegate this access right. Similarly, cert (3) grants the subject "K_B Carol" access to whatever K_B has access to. The ■ at the end of this cert indicates that the subject may not delegate this access right.

$$\begin{array}{c}(1)\ K_R\ \square \rightarrow K_A\ \text{Bob}\ \square \\ (2)\ K_A\ \text{Bob} \rightarrow K_B \\ (3)\ K_B\ \square \rightarrow K_B\ \text{Carol}\ \blacksquare \\ (4)\ K_B\ \text{Carol} \rightarrow K_C\end{array}$$

Fig. 2. SPKI/SDSI certificates as rewrite rules

The use of delegation and an indirect naming scheme means that more than one certificate may be necessary for a principal to access a resource. Such a set of one or more certificates is called a *certificate chain*. A certificate chain may also be represented by a rewrite rule, derived from the composition of compatible certificates. As defined in [2], certs $C_1 = K_1\ A_1 \rightarrow S_1$ and $C_2 = K_2\ A_2 \rightarrow S_2$ are *compatible* if $S_1 = K_2\ A_2\ X$ for some sequence of zero or more identifiers X (that is $K_2\ A_2$ is a prefix of S_1). The *composition* of C_1 and C_2, written as $C_1 \circ C_2$ is defined by replacing the prefix of S_1 with S_2. Using the term rewriting notation:

$$\begin{array}{c}C_1 = K_1\ A_1 \rightarrow K_2\ A_2\ \text{X} \\ C_2 = K_2\ A_2 \rightarrow S_2 \\ C_1 \circ C_2 = K_1\ A_1 \rightarrow S_2\ X\end{array}$$

Certificate chains are built by repeated use of composition.

Returning to the examples in Fig. 2, we can form cert chains by composing compatible certificates. $(1) \circ (2) = K_R\ \square \rightarrow K_B\ \square$ authorizes K_B to access resource R and to delegate that access right; $(3) \circ (4) = K_B\ \square \rightarrow K_C\ \blacksquare$ grants K_C access to whatever K_B has access to. Putting these two chains together we get the chain $((1) \circ (2)) \circ ((3) \circ (4)) = K_R\ \square \rightarrow K_C\ \blacksquare$ that authorizes K_C to access resource R, but not to delegate that access right. The problem of assembling such a chain is called the certificate chain discovery problem. Solutions based on formal language techniques can be found in [2, 11].

The ClassAd representation of SPKI/SDSI certificates is fairly simple. Each certificate ClassAd consists zero or more *cert request ports* and a *cert offer port*.

A cert offer port contains attributes corresponding to the type (name or auth), issuer, identifier (name certs only), and subject of the cert. The `Subject` attribute is a literal value if the subject of the cert is directly specified using a public key, or an attribute reference if the subject is indirectly specified using a SPKI/SDSI name with one or more identifiers. In the indirect case the ClassAd also contains one or more cert request ports, each of which requests a name cert (or chain of certs) to resolve the SPKI/SDSI name. If the ClassAd represents an auth cert with the delegation bit turned on, there is an additional cert request port requesting an additional auth cert (or chain of certs) issued by the subject of the cert.

For example, the authorization certificate designated as (1) in Fig. 2 would be represented by the ClassAd shown in Fig. 3. The name certificate designated as (2) in Fig. 2 would be represented by the ClassAd shown in Fig. 4.

```
[Ports = {
  [other = chain1; Type = "cert_request";
   Requirements = other.Type == "cert_offer" && other.CertType == "Name" &&
     other.Issuer == "K_A" && other.Identifier == "Bob";
  ],
  [other = chain2; Type = "cert_request";
   Requirements = other.Type == "cert_offer" &&
     other.CertType == "Auth" && other.Issuer == chain1.Subject
  ],
  [other = request; Type = "cert_offer"; CertType = "Auth";
   Issuer = "X"; Subject = chain2.Subject;
   Requirements = other.Type == "cert_request"
  ]}
]
```

Fig. 3. The ClassAd for cert(1)

```
[Ports = {
  [other = request; Type = "cert_offer"; CertType = "Name";
   Issuer = "K_A"; Identifier = "Bob"; Subject = "K_B";
   Requirements = other.Type == "cert_request"
  ]}
]
```

Fig. 4. ClassAd for certificate (2)

4 Gangmatching Structures and Concepts

As we have seen in the examples above, a gangmatching ClassAd is made up of a set of ports, each of which represents a request for another ClassAd. We

formally define a port P as a 5-tuple (E_P, I_P, J_P, δ_P, ϕ_P) where E_P is the set of all attributes defined or *exported* by P, I_P is the set of all attributes imported from the ClassAd that is matched with P, J_P is the set of all attributes referenced in P that are imported via other ports in the same ClassAd, δ_P is a function representing the attribute definitions in P, ϕ_P is a Boolean expression in disjunctive normal form (DNF) over I_P, J_P representing the `Requirements` expression of P. A ClassAd C is defined as an ordered list of ports.

The gangmatching process assembles a gang of ClassAds that is *complete* when all ports of all ClassAds in the gang have been matched with ports of other ClassAds in the gang. A *gangster* is an intermediate structure formed during gangmatching that represents an *open* or unmatched port in an incomplete gang. We define a gangster G as a triple (P, β, L) where P =(E_P, I_P, J_P, δ_P, ϕ_P) is a port, β is a function that binds the attributes in J_P to literal values, and L associates attributes imported from elsewhere in the gang with attributes imported from the ClassAd that will ultimately be matched with P. A port connecting a ClassAd C to one of its children is called a *child port*, and the port connecting C to its parent is the *parent port*. A gang can be thought of as a tree of ClassAds where each ClassAd is connected to its parent or child through one of its ports. The ClassAd at the root of the tree is referred to as the *root ClassAd*.

The gangmatching algorithm relies heavily upon the concepts of *equivalence*, *partial evaluation* and *validity*. Two gangsters are *equivalent* if they are structurally the same, but contain attributes from different ClassAds. An individual match is *conditionally valid* if one or both of the `Requirements` expressions involved unresolved attribute references. Partial evaluation is used to condense these expressions, which must then be satisfied by bindings generated by subsequent matches. A gang in which all of these expressions have been satisfied is considered a *valid* gang.

The input to the algorithm is the *root* ClassAd C_0 and a set of additional ClassAds $\mathcal{C}$ that will be used to build the rest of the gang. Beginning with the gangster consisting of the single port of C_0, the algorithm creates new gangsters by matching existing gangsters to parent ports of other ClassAds. Whenever a new gangster is created, a new rule in a regular grammar is generated. When the algorithm terminates, this grammar generates all complete valid gangs built from C_0 and the ClassAds in $\mathcal{C}$. In order to avoid repeated work and infinite loops caused by the reuse of ClassAds, the algorithm must test each new gangster for equivalence to previously encountered gangsters. If an equivalent gangster is found, the algorithm adds a new rule to the grammar, but does not attempt to match the new gangster. Otherwise, the new gangster is tested against the parent port of each ClassAd in $\mathcal{C}$ for a potential match. If the match is conditionally valid, the `Requirements` expressions of the respective ports are partially evaluated, and the resulting expression is passed to the first new gangster created by the match. Further matches must satisfy this expression in addition to the `Requirements` expressions of other ports encountered later.

The structures and concepts described here are examined in more detail in [4].

5 Gangmatching Algorithm

The gangmatching algorithm builds individual gangs in a top-down (root to leaves) fashion. The premise of the algorithm is that if an infinite number of gangs can be composed from a finite set of ClassAds, then there must be a repeating pattern – in the same way that a finite automaton can define an infinite but regular language. These repetitions can be prevented by detecting new gangsters that are equivalent to previously encountered gangsters. Thus, we can assemble a finite grammar that may produce an infinite number of gangs. In addition, this algorithm makes use of the partial evaluation facility described in Sect. 4 to build gangs that satisfy conditionally valid matches.

The algorithm takes as input a set $\mathcal{C}$ of ClassAds, and a root ClassAd C_0. Without loss of generality we will assume C_0 has only one port. We also assume that each ClassAd $C \in \mathcal{C} \cup \{C_0\}$ satisfies the following properties:

1. The `Requirements` expression ϕ_P of each port P of C consists of a conjunction of binary or unary predicates over attributes imported via P (I_P), attributes imported via previous ports in C (J_P) and literal values (represented by the set $\mathcal{V}$) in which no predicate contains attributes imported from more than one previous port in C and every predicate contains at least one attribute imported via P.
2. The last port in C is the parent port of C, and all other ports are child ports.
3. C has no more than 2 child ports.

In order to facilitate the handling of conditionally valid matches we will add an additional component ψ_G to each gang G. The purpose of ψ_G will become clear as we discuss the algorithm.

The following methods are not explicitly defined here:

1. AddGangster - adds a new gangster to a queue to be processed later
2. AddRule - adds a new rule to the grammar
3. MoreGangsters - returns **true** if more unprocessed gangsters are available, **false** otherwise
4. RemoveGangster - removes a gangster from the queue
5. CheckSeen - checks if a gangster is equivalent to a previously encountered gangster, and adds it to the previously seen gangsters if it hasn't
6. AddExtraRules - finds any rules containing a gangster equivalent to given gangster, and creates duplicates of those rules for the given gangster
7. MatchResults - tests a match between a gangster and a ClassAd, and returns an expression generated by partially evaluating and conjoining the `Requirements` expressions of the gangster and ClassAd
8. ValidMatch - determines if the result of a match indicates that it is valid (both `Requirements` expressions evaluate to true, or can be partially evaluated to satisfiable expressions)
9. SetNext - adds a link to a list of gangsters in an incomplete gang
10. GetNext - gets then next gangster in the list of gangsters.

The GangMatch method shown in Fig. 5 adds a gangster created from the single port of C_0. The algorithm then enters a loop in which gangsters are removed and added to a list of gangs using the AddGangster and RemoveGangster methods. At the beginning of each loop, a gangster G is selected and tested to see if an equivalent gangster has been previously encountered using the CheckSeen method. If CheckSeen returns **true**, the AddExtraRules method is called, adding new rules containing G to the grammar based on existing rules containing equivalent gangsters. If CheckSeen returns **false**, the ProcessMatch method is called on each $C \in \mathcal{C}$ to see if it matches G. When the GangMatch method has completed, the generated grammar will produce a set of matches representing all complete valid gangs rooted at C_0. Each gang is a list of ClassAds in order of appearance in the gang, with the parent port of each ClassAd matching the first open port of the gang made up of the previous ClassAds.

```
GangMatch(C0, C)
 1 P ← C0's port
 2 G ← (P,∅,∅,T)
 3 AddGangster(G)
 4 AddRule(G → C0)
 5 while MoreGangsters()
 6     G ← RemoveGangster()
 7     if CheckSeen(G)
 8         AddExtraRules(G)
 9     else
10         for each C ∈ C
11             ProcessMatch(G, C)
```

Fig. 5. The GangMatch algorithm

The ProcessMatch method shown in Fig. 6 tests the match between G and C using the MatchResults method. The ValidMatch method is then used on the resulting expression to determine whether or not the match was valid or conditionally valid (i.e. further matches will be needed). If ValidMatch returns **true**, the MatchBindings, ProcessPorts, and ProcessNextGangster methods are called to process any new gangsters generated by the match.

The MatchBindings method shown in Fig. 7 creates a set of bindings to be used by ProcessPorts and ProcessNextGangster. The bindings are produced using the set of attribute definitions δ_P contained in C's parent port P. If any attribute defined in δ_P $(attr, Y)$ corresponds to an attribute referenced in the set L_G of existing bindings in G $(X, attr)$, a new binding $(X, Y)$is created and added to the set L_M. Additionally, a binding is created from the attribute definition itself $(attr, Y)$. Once all attribute definitions in δ_P are checked, the set of bindings L_M is returned.

```
ProcessMatch(G, C)
1 ψ_M ← MatchResults(G, C)
2 if ValidMatch(ψ_M)
3     L_M ← MatchBindings(G, C)
4     G_last ← ProcessPorts(G, C, ψ_M, L_M)
5     ProcessNextGangster(G, C, L_M, G_last)
```

Fig. 6. The ProcessMatch method

```
MatchBindings(G, C)
1 P ← C's parent port
2 L_M ← ∅
3 for each (attr, Y) ∈ δ_P
3     if (X, attr) ∈ L_G
4         L_M ← L_M ∪ {(X, Y)}
5     L_M ← L_M ∪ {(attr, Y)}
6 return L_M
```

Fig. 7. The MatchBindings method

The ProcessPorts method shown in Fig. 8 goes through each port in C and creates a new gangster corresponding to that port based on the results of the match. The method takes as arguments G, C, the resulting expression ψ_M from the match between them, and the set of bindings L_M generated by MatchBindings. First, L_M is searched for any binding (X, Y) where Y is a member of the set I_P of imported attributes in P, and the resulting bindings are added to the set L. Second, psi_M is searched for any predicates containing an attribute in I_P, and the results are conjoined to form the expression psi. A new gangster G_{new} is then created from P, L, and psi, and is added to the queue of new gangsters. If there are no prior gangsters in the gang, a new rule $G_{new} \rightarrow G\ C$ is added to the grammar to indicate that G_{new} is a result of matching G and C. Finally, G_{new} is added to the linked list of gangsters comprising the current gang. The last gangster generated is returned by the method.

The ProcessNextGangster method shown in Fig. 9 updates the next gangster in the gang after G to reflect the results of the match between G and C. Like ProcessPorts the ProcessNextGangster method takes G, C, and L_M as arguments, along with the last gangster G_{last} created by ProcessPorts. The method begins by checking if there are any more gangsters in the gang after G. If there are no more gangsters, G_{last} is set as the last gangster in the gang. If there was no G_{last} the gang must be complete and the rule $S \rightarrow G\ C$ is added to complete the grammar. If there is a next gangster G' it must be updated.

The update of G' proceeds in a manner similar to the generation of new gangsters in ProcessPorts. First, L_M is searched for any binding (X, Y) where Y is a literal value, and X is a member of the set of attributes $J_{P_{G'}}$ imported in $P_{G'}$ from previous ports in the ClassAd containing $P_{G'}$. The resulting bindings

```
ProcessPorts(G, C, ψ_M, L_M)
 1 G_last ← null
 2 for each child port P of C
 3     L ← {(X, Y) ∈ L_M | Y ∈ I_P}
 4     ψ ← ∧ {preds in ψ_M over i ∈ I_P}
 5     G_new ← (P, ∅, L, ψ)
 6     AddGangster(G_new)
 7     if G_last = null
 8         AddRule(G_new → G C)
 9     else
10     SetNext(G_last, G_new)
11     G_last ← G_new
12 return G_last
```

Fig. 8. The ProcessPorts method

```
ProcessNextGangster(G, C, L_M, G_last)
 1 G′ ← GetNext(G)
 2 if G′ ≠ null
 3     β ← {(X, Y) ∈ L_M | X ∈ J_{P_G′}, Y ∈ V}
 4     G_new ← (P_G′, β, L_G′, ψ_G′)
 5     AddGangster(G_new)
 6     if G_last = null
 7         AddRule(G_new → G C)
 8     else SetNext(G_last, G_new)
 9     SetNext(G_new, GetNext(G′))
10 elsif G_last ≠ null
11     SetNext(G_last, null)
12 else
13     AddRule(S → G C)
```

Fig. 9. The ProcessNextGangster method

are stored in the set of bindings β, which is added to G' to create the new gangster G_{new}. The remainder of the method is similar to lines 7-10 in ProcessPorts in which the rule $G_{new} \rightarrow G\ C$ is added to the grammar if it is the first gangster in the gang, and the linked list of gangs is adjusted to include G_{new}.

6 Gangmatching Analysis

Gangmatching analysis is essentially an extension of bilateral matching analysis [3]. Between any two given ports, the same techniques can be used to determine why the first port does not match the second and vice versa. However, the presence of prior ports in a ClassAd introduces the possibility that one match may be dependent on the results of other matches. In addition, new problems arise from the more complex structure of a gang as opposed to two matching ClassAds.

A common problem in authorization systems is how to revoke a principal's access to a resource. For example, in SPKI/SDSI a principal may have access to a resource via several different certificate chains containing certificates issued by several different principals. In order to revoke the principal's access to the resource, at least one certificate in each such chain must be revoked. To avoid unnecessary disruption caused by certificate revocation, the set of certificates revoked should be minimal.

The *Break the Chain* problem may be abstracted to the problem of finding a minimal element in a subset lattice that passes a given test. In this case the top set in the lattice is the set of all certificates in C. The test on a given C' $\subseteq C$ is whether the certificates in C' grant the principal access to the resource. The problem of finding all such minimal elements has been shown to be NP-hard [10], but the problem of finding one such element is linear. Furthermore, finding k such elements for a constant k is polynomial: for $k > 1$ the complexity is $O(n^{k-1})$. The algorithm itself [4] applies this abstraction, then improves the performance by optimizing to reduce repeated work.

The *Missing Link* problem is the opposite of the Break the Chain problem. In this case a principal has no access to a resource, but may have elements of a certificate chain that would grant access. The problem is to find which certificates are needed to complete a chain that will authorize the principal to access the resource. The gangmatching equivalent of this problem is finding which ClassAds are needed to complete a gang. The solution to this problem is to run the gangmatching algorithm with a slight modification: When a port does not match any other ports, the gang is not abandoned; instead, the algorithm continues to match the rest of the ports in the gang and any dependencies on the unmatched port are ignored. When a partial gang has been completed, the "missing links" in the gang can be determined by using the `Requirements` expressions of the unmatched ports, and the references to imported attributes in these ports. Satisfied `Requirements` expressions elsewhere in the gang that contain such references can be partially evaluated to produce additional constraints for missing links. The gangmatching algorithm can be modified [4] to accept prototype ClassAds that will capture these additional constraints.

7 Related Work

There are some similarities between ClassAds and agent communication languages [9, 7, 17], though ClassAds employ a representation more akin to a database record than the rule-based representation used by these languages. There are also similarities between ClassAds matchmaking and the unification-based matching used by Linda [8] and Datalog. Linda uses tuples containing variables or literals to search a tuple space for a matching tuple. Datalog operates similarly on relational databases.

The term rewriting approach to SPKI/SDSI was introduced in [2] along with an algorithm for certificate chain discovery. It is also possible to use pushdown systems (PDS) to represent SPKI/SDSI rewrite rules [11, 12]. The enhanced

gangmatching algorithm in Sect. 5 began as a generalization of the *post** algorithm for PDS reachability.

The resource selection and authorization policies discussed in this paper both fall under the category of *provisions*. Provisions are conditions that must be satisfied or actions that must occur before a decision takes place. In contrast *obligations* are conditions or actions that must be fulfilled after a decision has been made [1]. An SLA is an agreement between a service provider and a customer that specifies certain attributes of the service such as availability, serviceability, performance and operation [19]. PDL [13] expresses obligation policies as event-condition-action rules. The Ponder policy language [5] can also be used to express both obligation and authorization policies.

Several other policy languages – such as Rei and Kaos have been developed specifically for the semantic web and grid computing applications. These languages are typically based on description logics such as DAML and OWL. A comparison of Rei, Kaos and Ponder is presented here [18].

8 Conclusions

Distributed computing environments provide users with a wide range of services that a single isolated system can not provide. Policies must be designed and enforced to protect the interests of users and providers of these services. Resource selection policies address the question: What kind of resource does a principal want, and is such a resource available? Access control policies address the question: Can a principal be trusted to have access to a given resource?

The framework for policy specification and interpretation presented in this paper provides a clearing house for both types of policies. It is built on the simple yet powerful concept of matchmaking. The ClassAd language and matchmaking algorithms were initially developed to solve resource selection problems in a distributed system. As we have shown, the same framework with some minor modifications is applicable to managing access control policies.

We have demonstrated that the ClassAd language can be used to specify SPKI/SDSI authorization policies, and an enhanced gangmatching algorithm can be used to assemble SPKI/SDSI certificate chains correctly and efficiently. We have also presented the necessary theoretical underpinnings of the enhanced gangmatching algorithm which generalize beyond the specific instance of SPKI/SDSI certificate chain discovery. Finally, we have demonstrated analysis techniques for bilateral and multilateral matchmaking that serve as essential tools for comprehending matchmaking results. Taken together these contributions provide a robust framework for specifying and interpreting resource allocation policies.

References

[1] Bettini, C., Jajodia, S., Wang, S., Wijesekera, D.: Provisions and obligations in policy rule management and security applications. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China, pp. 502–513 (August 2002)

[2] Clarke, D., Elien, J.-E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.: Certificate chain discovery in SPKI/SDSI. Journal of Computer Security 9(4), 285–322 (2001)
[3] Coleman, N., Raman, R., Livny, M., Solomon, M.: Distributed policy management and comprehension with classified advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin (April 2003)
[4] Coleman, N.: A Matchmaking Approach to Distributed Policy Specification and Interpretation. PhD thesis, University of Wisconsin-Madison (August 2007)
[5] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)
[6] Ellison, C., Frantz, B., Lampson, B., Rivest, R.L., Thomas, B., Ylonen, T.: SPKI certificate theory. RFC 2693 (September 1999)
[7] Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proc. of the Third Int'l Conf. on Information and Knowledge Management, CIKM 1994. ACM Press (November1994)
[8] Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
[9] Genesereth, M., Singh, N., Syed, M.: A distributed anonymous knowledge sharing approach to software interoperation. In: Proc. of the Int'l Symposium on Fifth Generation Computing Systems, pp. 125–139 (1994)
[10] Godfrey, P.: Minimization in cooperative response to failing database queries. International Journal of Cooperative Information Systems (IJCIS) 6(2), 95–149 (1997)
[11] Jha, S., Reps, T.: Analysis of SPKI/SDSI certificates using model checking. In: Proceedings of IEEE Computer Security Foundations Workshop (CSFW). IEEE Computer Society Press (2002)
[12] Jha, S., Reps, T.W.: Model checking spki/sdsi. Journal of Computer Security 12(3-4), 317–353 (2004)
[13] Lobo, J., Bhatia, R., Naqvi, S.: A policy description language. In: AAAI/IAAI, pp. 291–298 (1999)
[14] Raman, R., Livny, M., Solomon, M.: Matchmaking: Distributed resource management for high-throughput computing. In: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, HPDC7 (July 1998)
[15] Raman, R., Livny, M., Solomon, M.: Policy driven heterogeneous resource coallocation with gangmatching. In: Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC12), Seattle, WA (June 2003)
[16] Solomon, M.: The ClassAd language reference manual version 2.4 (May 2004), `http://www.cs.wisc.edu/condor/classad/refman/`
[17] Sycara, K., Decker, K., Pannu, A., Williamson, M., Zeng, D.: Distributed intelligent agents. IEEE Expert, 36–46 (December 1996)
[18] Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N., Uszok, A.: Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 419–437. Springer, Heidelberg (2003)
[19] Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J., Waldbusser, S.: Policy terminology. RFC 3198 (November 2001)