A Constraint Language Approach to Matchmaking

Chuang Liu[1] Ian Foster[1,2]
[1]Department of Computer Science, University of Chicago, Chicago, IL 60637
[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
{chliu, foster}@cs.uchicago.edu

Abstract

The need to discover and select entities that match specified requirements arises in many contexts in distributed systems. Meeting this need is complicated by the fact that not only may the potential consumer specify constraints on resources, but the owner of the entity in question may specify constraints on the consumer. This observation has motivated Raman et al. to propose that discovery and selection be implemented as a symmetric matching process, an approach they take in their ClassAds system. We present here a new approach to symmetric matching that achieves significant advances in expressivities relative to the current ClassAds—for example, allowing for multi-way matches, expression and location of resource with negotiable capability. The key to our approach is that we reinterpret matching as a constraint problem and exploit constraint-solving technologies to implement matching operations. We have prototyped a matchmaking mechanism, named *Redline*, and used it to model and solve several challenging matching problems.

1. Introduction

The development of Internet and Grid technologies [7] has led to a remarkable increase in the number of resources to which a user, program, or community may have access. The dynamic nature of distributed systems, however, means that these resources may appear and disappear unpredictably. Thus, we require scalable, efficient, and expressive mechanisms for the automated discovery and selection of resources that meet specified requirements. (Here, and in the rest of this article, we use the term *resource* as a generic term to indicate a physical device, service, data item, or other entity for which discovery and selection procedures are required. We believe that our techniques are broadly applicable.)

Complicating the resource selection problem is the fact that in many situations the resources that we seek to discover may themselves place requirements on acceptable requests. For example, the autonomous nature of Grid resources may result in a resource allowing access only to users belonging to a certain group or able to pay a fee. This observation led Raman et al. [22] to propose that resource search and selection be treated as a bilateral *matching* process. In their approach, properties of requests and resources are characterized in a common syntax capable of representing both attributes and requirements. A symmetric *matching* step is then used to determine, for a particular request-resource pair, whether requirements are mutually satisfied.

While ClassAds mechanism has been applied successfully in numerous application domains, its syntax limits its usage on location of resources with negotiable and adaptable properties. In this article, we report on work that extends the power and scope of the matchmaking concept significantly by treating matching as a constraint problem. We describe a new matching system within which we can do the following, none of which are supported within ClassAds.

- *Describe and match resources whose properties are expressed by sets or ranges.* ClassAds describe capability of a resource by binding a value to a property. Yet some resources have complex properties that cannot be described by a single value. For example, a network resource might providing a 155 MBytes/s connection for $20/hour, a 622 MBytes/s connection for $100 /hour, and a 2.4 GBytes/s connection for $300/hour between two sites. We design syntax and a matching mechanism to describe and located resources whose properties are expressed by sets or ranges.
- *Provide useful matching result.* ClassAds matchmaker returns only binary result matched or unmatched. Our matchmaker instead can point out the unsatisfied requirements if a match fails. Furthermore, some resources have properties or access policies that are adaptable or negotiable. The matching result should contain the actual properties that enable the successful match. For example, a resource owner might allow other users to access his resource from 7:00 to 10:00 PM. For a request "Find a machine that allows access between 8:00 PM and

11:00 PM," our matchmaker will return that this resource is available from 8:00 PM to 10:00 PM.

- *Support matching between multiple requests and a resource*. We define this kind of match as *congruous match*. For example, a group wants to add a new machine and ask its members for suggestions. Because every member has its own requirements for this resource, we need the matchmaker to check if there is congruence among these requests and locate a resource that can satisfy everybody. In this scenario, a match happens between multiple requests and a resource. ClassAds perform only the match between one request and one or multiple resources. Our matching mechanism can handle all these kinds of matches in a general way.
- *Support matching between a request and a resource set with particular aggregated properties*. We define this kind of match as *set match*. An example of requests is to find a set of computers with total memory size bigger than 10G. The latest version of ClassAds is extended to support set match based on our experience.

To permit experimentation with our approach, we have designed and prototyped a language and a matching mechanism that we collectively called *Redline* named after a train route in Chicago. We have applied *Redline* to some challenging matching problems from several application domains.

The rest of this paper is structured as follows. Section 2 describes related work. Section 3 describes the description language. In section 4, we illustrate our matching mechanism by four matching problems. We conclude and outline our plans for future work in Section 5

2. Related work

The matching problem occurs in many contexts, and we find a wide variety of approaches to its solution. Here we review the most relevant previous work, focusing in particular on research within distributed computing and e-commerce.

Information systems. Much effort has been devoted to developing *information systems* for publishing, aggregating, and supporting queries against collections of resource descriptions (SNMP [23], LDAP [11], MDS [4], UDDI [16]). Such systems differ in various dimensions, such as their description syntax (e.g., MIBs [23], relations [8], LDAP objects [11]), query language (e.g., SQL [8], Xquery [3], LDAP query [11]), and the techniques used to publish and aggregate resource descriptions (e.g., soft state vs. stateful servers, complete descriptions vs. Bloom filters). However, these systems all have in common that the resource provider-consumer interaction is *asymmetric*: the provider-published description of its properties is queried by the consumer to identify candidates prior to generating requests for resources. The consumer selection procedure may comprise a simple query or, alternatively, involve a procedural algorithm based, for example, on a performance model [2,5].

Although these information systems provide access control based on users' accounts/IDs, we argue that it are not enough for express and enforce resource access policy, which restricts the availability of resource based on not only requesters' accounts/IDs, but resource access time and usage of this resource etc. Furthermore, data access controls in these information systems are configured manually by an administrator. Considering the numerous numbers of resources and their heterogeneous policies in an information system, it won't be an easy work for an administrator to manage them. Thus we model resource selection as a *symmetric* process that enables the easy expression and enforcement of resource provider policy.

Symmetric evaluation. Symmetric evaluation was pioneered by the Condor *matchmaker* system [12], in which both requests and descriptions are expressed using the same ClassAds syntax. A ClassAd can contain (a) properties (of a request or resource), expressed as attribute/expression pairs; (b) requirements that must be satisfied by a matching ClassAd, expressed as a Boolean *requirements* statement; and (c) a function used to assign a numeric rank to a matching ClassAd, expressed as a *rank* statement. Two ClassAds match if the *requirements* expression of each evaluates to true.

```
Request = [  user = "chliu";
          requirements = other.type=="machine"
             && other.cpuspeed > 500M;
          rank = other.memsize ]
Resource = [ name="foo"; type="machine";
          cpuspeed=800M;  memsize=512M;
          requirements= DayTime() > '18:00'
             && member(other.user, {"chliu","lyang"})]
```

Figure 1. Two examples of Condor ClassAds. See text for details.

ClassAds uses *requirements* expression in request description to describe requirements to the resource and *requirements* expression in resource description to describe its policy. Thus a success match between a resource and a request means this resource has required properties and is accessible for this request. Figure 1 shows two example ClassAds. The first, `Request`, describes a request with a single property `owner=chliu`, a *requirements* statement requesting a computer with a CPU faster than 500 MHz, and a *rank* statement that returns the memory size. (The syntax `Other.attr` here is used to denote the value of attribute `attr` in the other ClassAd.) The second ClassAd, `Resource`, describes a computation resource named "foo" with an 800 MHz CPU and 512 MBytes memory, and with requirements

indicating that it is accessible only to users "chliu" and "lyang" and only after 6:00 PM.

The Condor equivalent of an information system such as UDDI is a matchmaker that maintains a pool of ClassAds representing candidate resources and then matches each incoming request ClassAd with all candidates, returning a highest-ranked matching candidate.

Other symmetric matchmaking systems used for resource selection include the Jini lookup service [1], DAML-S matchmaker [18], LARKS [24], and the HP e-commerce matchmaker [9,20]. These systems use a range of different syntaxes (Java object, service profile, RDF, description logic concept) and matching mechanisms (semantic or syntactic), but have in common the use of symmetric evaluation mechanisms in which resource and request are described by the same syntax, and a matchmaker checks if these two descriptions match each other.

However, these systems have limitations when it comes to expressing and locating resources with property values that are a feature set or range, as suggested by IETF RFC2506 [10] and W3C CC/PP [17], to locating a resource compatible for a set of congruent requests, and to expressing and locating multiple resources with a particular relationship: the *resource co-selection* problem, as discussed by Dinda [6,19], Raman [21], Czajkowski [4], and Liu [14], among others.

We argue that no previous work solves these problems completely. We present a solution to overcoming these limitations based on the use of a constraint programming language approach to modeling and implementing the matchmaking process.

3. Description of resources and requests

Existing resource information systems and matchmaking systems usually describe resource properties by attribute value pairs, while describe a request by constraints on attributes that specified possible satisfying values. A match happens if a resource falls in the range specified by constraints in a request. Resources with adaptable/negotiable properties bring challenges to this scheme because it is hard to use attribute/value pairs to describe these properties. For example, for a network resource that provides different network connections at different prices, we cannot describe its capability by simply binding a value to an attribute `bandwidth` because this attribute may have different values based on the price a user wants to pay. In this scenario, a resource needs to specify a finite value set or range that an attribute can take, and a match happens if there exists intersection between resource properties and required properties in a request. Based on this observation, we extend the original scheme by using constraints on attributes to describe properties of both resources and requests. Also, by using constraints, a resource is able to specify its access policy, which specifies the range of allowing requests, in its description.

Definition 1: A *constraint* C is a relationship that must be hold when choosing value for all attributes occurring in this constraint. For example, constraint `cpu > 500` means attribute `cpu` is a value bigger than 500.

We develop a description language called *Redline* to express requests and resources. Its syntax is similar to constraint programming languages. Different from other constraint languages that are usually used to model a problem, *Redline* introduces new data types and constraints that make it suitable as a resource/request description language.

In *Redline*, data types include basic types (such as *integer, real, string* etc.), composite data type *set*, and structure data type *description*. *Description* is a set of constraints in the format of `[<constraint 1>; <constraint 2>; …; <constraint n>]`. It may be used to represent a resource, request, or a feature with multiple attributes. *Set* is an aggregation of values with the same type that can be any basic types or *description*. The syntax of *Set* is `[<value 1>, <value 2>, …, <value n>]`. Several set-related functions are provided to get the aggregation characteristics of a set, such as `Sum(<set>)` that calculates the sum of all values in <set>.

```
1.  N1 = [type="network";
2.          From="s1.uchicago.edu"; To = "m2.ucsd.edu";
3.          connections = ENUM [
4.                  [ band = 155; cost = 20],
5.                  [ band = 622; cost = 100],
6.                  [ band = 2400; cost = 300] ] ]
7. N2 =[  type="network";
8.          From="s1.uchicago.edu"; To = "m2.ucsd.edu";
9.          connections =  [ band = 155; cost= 20] ];
10. R = [  type="network"; connections.band > 155;
11.         From="s1.uchicago.edu"; To="m2.ucsd.edu"]
```

Figure 2. An example of network resource description

Constraints supported by *Redline* includes arithmetic functions (like =, +, and ×, etc.), logic functions (like >, <, etc.), *ISA, ISASET* and *=ENUM*. A constraint `<attr> ISA <description>` means value of `<attr>` is a description that matches `<description>`. A constraint `<attr> ISASET <description>` means value of `<attr>` is a description set with every element matching `<description>`. Definition of match is given in section 4. *=ENUM* is a constraint in the format of `<attr> =ENUM <set>` that means value of `<attr>` is limited to the value in `<set>`. We will use examples to illustrate the use of these constraints. For detailed syntax information, please refer to [13]. Because these constraints can be combined to construct complex constraints, for the purpose of clarity, we call them *primitive constraints.*

Figure 2 shows examples of resources and requests described by *Redline*. Description `N1` represents a network resource capable of providing a 155 MBytes/s network connection for $20/hour, 622 MBytes/s for $100/hour, or 2.4GBytes/s for $300/hour between two machines. Description `N2` represents a network resource capable of providing a 155Mbytes/s network connection for $20/hour. Description `R` asks for a network resource that has more than 155Mbytes/s bandwidth.

In this example, we use attribute `connections` to represent the capability of this network resource. Line 4-6 is a description set that describes a network link feature set provided by this resource. In line 3, the new constraint *=ENUM* constrains value of attribute `connections` to one of three features. We use attribute `band` to represent the bandwidth and use `cost` to represent the price for this resource. This example shows how easily we can describe resources whose property is a feature set. ClassAds can't express this kind of resources.

Because description itself is a data type in *Redline* syntax, it can be arbitrarily nested, leading to a natural way to express a feature set as shown in lines 4-6 of Figure 2 or a request for multiple resources as shown in Figure 3.

Request

```
1.  request= [user="globus-user";
2.    userGroup="dsl-uc";
3.    computation ISA
4.      [type="computation"; cpuspeed > 150; actime >18];
5.    storage ISA
6.      [ type="storage"; space > 100; actime > 18];
7.    storage.domain = computation.domain]
```

Resources

```
C= [type="computation"; hn="c1.uchicago.edu";
     cpuspeed=200;
     domain = "uchicago.edu"];
     accesstime > 17 ]
S= [type = "storage"; hn="s2.uchicago.edu";
     space=200; domain="uchicago.edu"]
```

Figure 3. An example of requests for multiple resources

The first part of Figure 3 is a *Redline* description specifying a request for two resources located in the same site (assume resources from the same site share a domain name): a computation resource with CPU speed faster than 150 MHz, a storage resource with space bigger than 100 G. The access time to these resources is after 6:00PM. The second part of Figure 3 shows two examples of resource descriptions: computer `C` and storage systems `R`.

In this example, we use attribute `computation` to represent the required computation resource. Required properties of this computation resource are described by a description type value in line 4. In line 3, the new constraint *ISA* constrains value of attribute `computation` to be a description that matches the description in line 4, or in the other words, has all required properties described in line 4. In the same way, we describe the required storage resource in line 5-6. The value of an attribute in a description can be accessed by operator "**.**". In this example, we use the constraint `storage.domain = computation.domain` to specify the requirement that these two resource are located in the same site.

4. Matching

In a grid environment, requesters and resource owners describe resources they want or provide. Matching is a process to check if a resource and a request are compatible to each other. Requesters and resource owners are required to use same attribute names to express resource properties and associate common meaning to responding values. For example, in Figure 3, both the requester and the resource owner use the variable `cpuspeed` to express CPU speed of a computation resource and MHz as a common integer unit.

By the syntax described above, we can express resources with adaptable or negotiable properties. In this part, we will introduce how to mach this kind of resources with requests. First, we will illustrate the flexibility of our matchmaking mechanism by using it to model four different types of matching problems. Then, by formalizing these matching problems into a constraint problem, we build a general matchmaking algorithm.

4.1. Matching problems

We define four typical types of matching problems based on the number of involved resources or requests. Bilateral matching and gang matching problem are first defined by Raman [21] in ClassAds system. We use a new method to model these two problems to enable matching of adaptable or negotiable resources. We also define congruence matching and set matching problem that haven't be modeled by current ClassAds system and suggest a solution to it.

4.1.1. Bilateral matching *Bilateral matching* problem is defined as: given a request and a resource, check if they match each other. Bilateral matching happens when a requester tries to locate a resource with particular properties. Usually, requesters use constraints to describe a range of resources that satisfy his requirements. By the syntax described in section 3, *Redline* also allows resource owners to describe a range of properties that a resource can provide. A request matches a resource if there exists an intersection between the range of required properties and the range of provided properties in two descriptions.

We use *Redline* description to describe resources and requests. A *Redline* description is a self-consistent collection of constraints over named attributes of an entity. We express a description as a constraint in the form of $D = c_1 \wedge \ldots \wedge c_n$, where n >= 0 and $c_1, \ldots, c_n$ are constraints on attributes. The symbol $\wedge$ denotes *and*.

Given a resource description D_1 and a resource description D_2, we formalize bilateral matching problem as a constraint $D_1 \wedge D_2$. We define that two description matches each other if this constraint is satisfiable. The algorithm to test constraint satisfiability is given in section 4.2.

Definition 2: A constraint C is *satisfiable* if there exists a value assignment to every variable $v \in vars(C)$ such that C holds. Otherwise, it is *unsatisfiable*. vars(C) denotes the set of variables occurring in constraint C.

For the example in Figure 2, matching N_1 with R is represented by a constraint:

type="network"
$\wedge$ From="s1.uchicago.edu"
$\wedge$ To="m2.ucsd.edu"
$\wedge$ connections=ENUM[…]
$\wedge$ type="network"
$\wedge$ connections.band>155
$\wedge$ From="s1.uchicago.edu"
$\wedge$ To="m2.ucsd.edu".

N_1 and R match each other because there exists an assignment for this constraint.

type="network";
From="s1.uchicago.edu";
To="m2.ucsd.edu";
connections=[band=622; cost=100]}

4.1.2. Gang matching. *Gang matching* problem is defined as: given a request for multiple resources that describes the required properties for every resource and their relationship, and a set of resources, check if these resources satisfy the request.

When describing a request for multiple resources by *Redline* syntax, we associate a description type attribute with every required resource and express requirements to this resource by constraint `ISA` (see example in Figure 3). A constraint `<attr> ISA <description>` is satisfied if the value of `<attr>` matches `<description>` based on the definition of bilateral match.

Given a request D that uses attributes V_i (i=1 to n) to represent required resources and a set of resource description D_i (i=1 to n), we formalize the gang matching problem as a constraint problems $D \wedge V_1=D_1 \wedge V_2=D_2 \wedge, \ldots, \wedge V_n=D_n$, We define D matches $D_1, D_2, \ldots, D_n$ if constraint $D \wedge V_1=D_1 \wedge V_2=D_2 \wedge, \ldots, \wedge V_n=D_n$ is satisfiable. For the example in Figure 3, matching request with resource R and S is represented by: `request ∧ computation=R ∧ storage=S`.

4.1.3. Congruence matching. *Congruence matching* problem is defined as: given a set of requests and a resource, a match succeeds if there is congruence among these requests and this congruence matches the resource.

An example of congruence matching is the resource location problem for a group mentioned in section 1. Figure 4 shows a simplified example. Three subgroups describe their requirements for a resource by D_1, D_2, and D_3 respectively. The group leader has a budget constraint on this resource described by D_4. We need to check if there exists congruence solutions. If yes, find the resource that can satisfy everybody. In Figure 4, D_1 describes a request for a machine with 4 processors, every processor faster than 500MHz and local disk space bigger than 10G. D_2 describes a request for a "linux" or "unix" machine with processor faster than 1000MHz, and memory bigger than 1000Mbytes. D_3 describes a request for a "linux" or "unix" machine with local disk space bigger than 128G. D_4 put a constraint on price of this resource: less than 1000$/hour. D shows a resource description for a machine with 1000MHz processor, 1200Mbytes memory size, 20G local disk size, and price is 500$/hour.

```
D1 = [ cpuspeed > 500; numofcpu > 4; localdisk > 10]
D2 = [ cpuspeed > 1000; memory > 1000;
       os =ENUM ["linux", "unix"] ]
D3 = [ localdisk > 128; os = ENUM["linux", "unix"]]
D4 = [ price < 1000]
D = [ cpuspeed = 1000; memory = 1200; os = "unix";
      localdisk = 20; price = 500]
```

Figure 4. An example of congruence matching

For a congruence matching problem with n requests, we use descriptions $D_1, D_2, \ldots, D_n$ to express requests and D' to express the resource. The congruence checking is formalized by a constraint $D = D_1 \wedge D_2 \wedge \ldots \wedge D_n$. Matching between requests and resource is formalized by a constraint $D \wedge D'$. A congruence matching succeeds if these two constraints are satisfiable. For the example above, congruence checking is represented by a constraint $D_1 \wedge D_2 \wedge D_3 \wedge D_4$.

4.1.4. Set matching. *Set matching* problem is defined as: given a request for a set of resources with particular aggregated properties and a set of resources, a match succeeds if this resource set has the required aggregated properties. An example of requests is "find a set of computers with total memory size bigger than 10G".

When describing a request by *Redline* syntax, we associate an attribute with required resource set by constraint `<attr> ISASET <description>`. Here `<description>` describes required properties for every resource in this set. We use set related functions to specify the requirement to aggregation properties of this set. For example, `[ cset ISASET [type="computation"; cpuspeed>100]; Sum(cset.mem) >10]` describes a

request for a set of computation resources with total memory size bigger than 10G and every resource should be faster than 100MHz.

Given a request D that uses attribute V to represent required resource set and a set of resource description D_i (i=1 to n), we formalize set matching problem as a constraint problems D ∧ V=[D_1, D_2,…, D_n]. We define D matches D_1, D_2, …, D_n if this constraint is satisfiable.

Although we only model four matching problems here, we believe *Redline* can be easily used to model other matching problems

4.2. Matchmaking algorithm

By formalizing all these matching problems into a constraint problem, we define that a match succeeds if the corresponding constraint problem is satisfiable. In this part, we introduce the matchmaking algorithm we implemented by extending existing constraint-solving algorithms.

Based on the type of constraints supported, different constraint-solving algorithms have been developed in constraint programming area, such as Gauss-Jordan elimination for linear constraint, backtrack algorithm for constraint problem with finite domain etc. These algorithms solve constraint satisfaction problem by finding an assignment to all variables without violating constraints.

In matchmaking scenario, however, we don't really need to get a particular value for every attribute. Some times, we even prefer not to decided value for every attribute because it will lose information. For example, for a machine only accessible to users from 7:00 to 10:00 PM and a request for a machine accessible between 8:00 PM and 11:00 PM, we describe this request and this resource by descriptions `[accesstime>19; accesstime<22]` and `[accesstime>20; accesstime<23]` separately. For the matching of these two descriptions, we prefer the matching result to be a range between 8:00PM and 10:00 PM, which is the accessible time for this user, than a particular value in this range. Also, for a failed match, users will be interested in getting information about what causes the mismatch. Based on this observation, we instead develop a matching algorithm that tries to find assignments to every attribute with finite value domain and constrain value ranges of attributes with infinite value domains as small as possible. Besides *matched* and *unmatched*, this algorithm outputs the information of attributes after combining all information described by constraints.

As mentioned in section 4.1, four matching problems are expressed by a constraint problem, which consists of a set of primitive constraints connected by ∧. The matching algorithm determines satisfiability of a constraint problem by testing consistency of these primitive constraints. This algorithm, as shown in Figure 5, comprises three steps.

```
// C is the constraint
// c is primitive constraints in C
1  Matchmaking_Algorithm(C)
2    For every primitive constraint c ∈C
3       b = NodeConsistency(c)
4       If (!b)       // inconsistency is found
5          Return unmatched
6       Endif
7    Endfor
8    b = LocalPropagation(C)
9    If(!b)    // inconsistency is found
10      Return unmatched
11   b= HyperArcConsistency(C)
12   If(!b)    // no matching state is found
13      Return unmatched
14   Return matched
```

Figure 5. Matchmaking algorithm

In the first step (lines 2-7), the matching algorithm repeatedly uses primitive constraints with one attribute to refine the information of this attribute and check for inconsistency, called node consistency algorithm [15]. If inconsistency is found, it returns *unmatched.*

In *Redline*, five types of primitive constraints are defined: arithmetic expression, logic expression, *ISA, ISASET* and *=ENUM*. An arithmetic expression with one attribute determines the value of this attribute; a logic expression with one attribute constrains the value of this attribute to a range; constraint *=ENUM* constrains the value of an attribute to a finite domain; and constraint *ISA* and *ISASET* are skipped and will be checked in the second step. Please note that nested description value is considered as a constant in this step. For example, an assignment constraint `<attr>=<description>` is considered as a constraint with only one attribute `<attr>`.

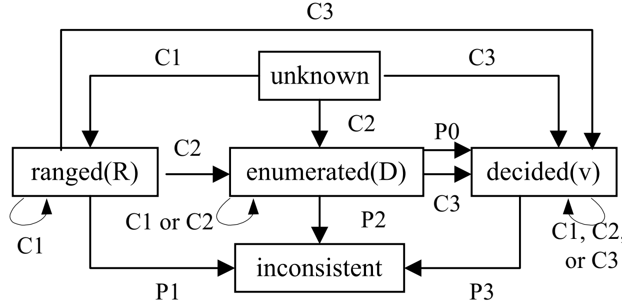
Based on information known about an attribute, an attribute has five states: *unknown* that means there is no information about this attribute, *decided* that means its value is determined, *ranged* that means its value is constrainted to a range, *enumerated* that means its value is in a finite domain, and *inconsistent* that means there is conflict information about this attribute. Initially, all attributes are in *unknown* state. We apply constraints on attributes and cause the transformation of states of attributes. The state transformations are shown in Figure 6.

In Figure 6, we use V to express a particular value, R to express a range, and D to express a finite domain. An attribute state is described in the format of <state name>(<information>). For example, ranged(R) means this attribute is in *ranged* state and its range is R.

As shown in Figure 5, the state of an attribute will change when a constraint is applied, and the resulting state is the combination of the information in the original state and in the constraint. For example, an attribute in the

ranged state will change to the *enumerated* state if applying a constraint *=ENUM* on this attribute (expressed by C2 in Figure 5). The domain D in the resulting state consists of elements that are specified in the constraint and are located in the range of the original state.

The state may also change if a particular condition is satisfied. For example, if the cardinality of a domain is empty (expressed by P2 in Figure 6), the state of the attribute is changed to inconsistent.



C1 expresses a logic expression that constrains value of the attribute to a range; C2 expresses a =ENUM constraint that constrains value of the attribute to a finite domain; C3 expresses an assignment constraint that binds the attribute to a particular vlaue.
P0 means that the cardinality of D is equal to 1; P1 means that R is empty; P2 means the cardinality of D is 0; P3 means that V is bound to two different values or a noexisted value.

Figure 6. Attribute state transformation

Because *Redline* allows arithmetic expressions and logic expressions with multiple attributes as constraints, we use local propagation [15] to simplify these constraints in the second step (lines 8-10 in Figure 5). This algorithm works by repeatedly selecting an attribute whose value is determined and replacing this attribute with its value in all constraints with multiple attributes. If a primitive constraint has only one attribute after the replacement, the node consistency algorithm is used to refine the information of this attribute. If any primitive constraint is evaluated to false in this step, the matching algorithm returns *unmatched.* For attributes with description type values, the related *ISA* and *ISASET* constraints are checked.

In the third step, we use the hyper-arc consistency algorithm [15] to check all possible assignments to attributes in the *enumerated* state and pick out those assignments causing no inconsistency (lines 11-13 in Figure 5). When a valid assignment is found and no attribute is in the *inconsistency* state, attribute states are remembered as a *matching state.* Unlike other constraint languages that only allow integer values to be enumerated, we support structure data types. In Figure 2, we use enumerated descriptions to express the available features. After three steps, if no matching state is found, matching algorithm returns *unmatched.* Otherwise, it returns *matched.*

Now we use the example in section 4.1.1 to illustrate the matching process. In this example, all primitive constraints in this constraint problem involve only one attribute. So we will apply them in the first step. For example, constraint `type="network"` will determine the value of attribute `type` to be a string `"network"`. So after applying this constraint, state of attribute `type` is *decided* with value `"network"`. Based on the tranformation rules (Figure 6), a following constraint `type="network"` causes no inconsistency and the state of attribute `type` won't be changed. Constraint `connections =ENUM [[band=155;cost=20], …, [band=2400;cost=300]]]` will constrain the value of attribute `connections` to one of these three descriptions. So after applying this constraint, state of attribute `connections` is *enumerated.* A following constraint `connections.band>155` will refine the domain of this attribute by removing value `[band=155;cost=20]` from its domain because this value is not located in the range described by this constraint. Because there is no constraint with multiple attributes, so we go directly to the third step after the first step. Here we need to check every possible value of `connections`. Because an assignment `connections=[band = 622; cost = 100]` causes no conflict, sate of attributes after this assignment is a matching state. Also an assignment `connections = [band=2400; cost=300]` causes no conflict. Now we can conclude that these two descriptions match each other and there are two matching states with `connections` pick different values.

Different from other matchmaking mechanism, this matchmaking algorithm outputs not only *matched* or *unmatched*, but also the state of attributes. For example, for the congruence matching problem in section 4.1.3, state of attribute `cpuspeed` will be a range from 1000MHz to infinite, which is a result of applying constraint `cpuspeed > 500` in D_1 and `cpuspeed > 1000` in D_2, after congruence checking. For a failed match, the attribute cause conflict will be marked by its state *inconsistent.* When there are multiple matching states, *Redline* allows requesters to control the returned state by a constraint `Minimize(attr)`, which instructs the matching algorithm to return the matching state that minimizes the value of `attr`. For the example in section 4.1.3, if the requester adds a constraint `Minimize(connections.price)`, only the matching state with `connections=[band = 622; cost = 100]` will be returned because cost value in this description is smaller.

5. Summary and future work

Resource selection in Grid environments usually involves multiple resources and requests with diverse ownership and policies. We have designed and implemented a matchmaking language, *Redline*, for expressing constraint associated with resources and request. We have also implemented a matchmaking process that uses constraint-solving techniques to solve the different kind of matching problem in a general way. The resulting system has significantly enhanced expressivities compared with previous approaches, being able to express resource with adaptable and negotiable properties and deal with resource selection operation that involve multiple resources and/or requests. *Redline* functions can be applied in a number of settings, including Globus Toolkit-based Grids and Web service directories.

Our current work is focused on evaluating the effectiveness of the *Redline* system in a wide range of applications; completing construction of a *Redline*-based resource selection service by designing the service interface and studying the organization of descriptions in the matchmaker.

Acknowledgments

This work was supported by the Grid Application Development Software (GrADS) project of the NSF Next Generation Software program, under Grant No. 9975020. We are grateful to Lingyun Yang and Alain Roy for comments on a draft of this paper.

References

[1] Arnold, K., Wollrath, A., O'Sullivan, B., Sheifler, R. and Waldo, J., *The Jini Specification*, Addison-Wesley, MA, USA, 1999.

[2] Berman, F. and Wolski, R., The AppLeS project: A Status Report. *Proceedings of the 8th NEC Research Symposium*, Berlin, Germany, 1997.

[3] Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J. and Siméon, J., XQuery 1.0: An XML Query Language. W3C, 2002.

[4] Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C., Grid Information Services for Distributed Resource Sharing. *10th IEEE International Symposium on High Performance Distributed Computing, Aug 7-9 2001*, Institute of Electrical and Electronics Engineers Inc., San Francisco, CA, 2001, pp. 181-194.

[5] Dail, H., A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. *Computer Science*, University of California, San Diego, 2002.

[6] Dinda, P. and Plale, B., A unified relational approach to grid information services. Grid Forum Informational Draft GWD-GIS-012-1, 2001.

[7] Foster, I. and Kesselman, C., *The Grid: Blueprint for A New Computing Infrastructure*, Morgan Kaufmann Publishers, San Francisco, 1999, xxiv, 677 pp.

[8] Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database Systems: the Complete Book*, Prentice Hall, Upper Saddle River, NJ, 2002, xxvii, 1119 pp.

[9] Gonzalez-Castillo, J., Trastour, D. and Bartolini, C., Description Logics for Matchmaking of Services. HP labs, 2001.

[10] Holtman, K., A. Mutz and Hardie, T., RFC 2506: Media Feature Tag Registration Procedure. 1999.

[11] Howes, T., Howes, T.A., Smith, M.C. and Good, G.S., *Understanding and Deploying LDAP Directory Services*, 2nd edn., Addison Wesley Professional, 2003, 608 pp.

[12] Litzkow, M., Livny, M. and Mutka, M., Condor - a hunter of idle workstations. *Proceedings of the eighth International Conference on Distributed Computing Systems*, 1988, pp. 104-111.

[13] Liu, C. and Foster, I., A Constraint Language Approach to Grid Resource Selection. University of Chicago, Chicago, 2003.

[14] Liu, C., Yang, L., Foster, I. and Angulo, D., Design and Evaluation of a Resource Selection Framework. *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, 2002.

[15] Marriott, K. and Stuckey, P.J., *Programming with Constraints: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.

[16] Newcomer, E., *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, Boston, 2002, xxviii, 332 pp.

[17] Nilsson, M., Hjelm, J. and Ohto, H., Composite Capability/Preference Profiles (CC/PP): Requirements and Architecture, *W3C Working Draft* (2000).

[18] Payne, T.R., Paolucci, M. and Sycara, K., Advertising and Matching DAML-S Service Descriptions. *International Semantic Web Working Symposium(SWWS)*, Stanford University, California, USA, 2001.

[19] Plale, B., Dinda, P. and Laszewski, G.V., Key Concepts and Services of a Grid Information Service. *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*, 2002.

[20] Preist, C., Agent Mediated Electronic Commerce at HP Labs, Bristol. Hewlett-Packard Labs, Bristol, 2001.

[21] Raman, R., Matchmaking Frameworks for Distributed Resource Management. *Computer Science*, University of Wisconsin, Madison, 2000.

[22] Raman, R., Livny, M. and Solomon, M., Matchmaking distributed resource management for high throughput computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, 1998.

[23] Stallings, W., *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, 3rd edn., Addison-Wesley, Reading, Mass., 1999, xv, 619 pp.

[24] Sycara, K., Widoff, S., Klusch, M. and Lu, J., LARKS: Dynamic Matchmaking Among Heterogeneous Software Agent in Cyberspace, *Autonomous Agents and Multi-Agent Systems* (2002) 173-203.