A Constraint Language Approach to Grid Resource Selection

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 expressivity relative to the current ClassAds—for example, allowing for multiway matches, expression and location of resource with negotiable capability and querying on policy. The key to our approach is that we reinterpret matching as a constraint satisfaction problem and exploit constraint-solving technologies to implement matching operations. We have prototyped a system that implements these ideas, *RedLine,* and successfully applied this prototype to some challenging matching problems from several application domains. This work both introduces an interesting new application for constraint language technologies and presents new ideas concerning the instantiation and implementation of those technologies in an application-specific setting.

1 Introduction

The development of Internet and Grid technologies [10] 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. [27] 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 policies. A symmetric *matching* step is then used to determine, for a particular request-resource pair, whether policies are mutually satisfied. This matching step can be embedded in a larger search as follows: (1) resource owners advertise their resources and access policies to a matchmaker, (2) the matchmaker stores these advertisements, (3) resource requesters advertise their resource requirements to the matchmaker, and (4) the matchmaker matches a request against resource advertisements and returns the result, a subset of the stored advertisements. Raman et al.'s implementation of this concept, the Condor matchmaker [20,26,27], has been applied successfully in numerous application domains.

- In this article, we report on work that extends the power and scope of the matchmaking concept significantly by treating matching as constraint satisfaction 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 property that can't be described by a single value. For example, for a screen capable of displaying 640*480 pixels and 16 million colors, 800*600 pixels and 64 thousand colors, or 1024*768 pixels and 256 colors, its property *resolution* is a feature set. We design syntax and matching mechanism to describe and locate resource whose properties are expressed by sets or ranges.
- *Match advertisements based on policy as well as properties*. ClassAds encode policies in requirements statements that cannot be queried. Yet policy may often form an important resource selection criterion. For example, a user may ask: "Find all machines that allow access between 7:00 PM and 9:00 PM." We allow requirements to be matched in the same way as other properties.
- *Matching resource sets as well as individual resources*. ClassAds perform only one-to-one matches. Yet, for example, a user may require a "set of computers, all faster than 800 MHz, with aggregate memory 10 Gbyte." We support such multiway matches, allowing a user to locate a collection of computers that meet our example requirement in the aggregate.

To permit experimentation with our approach, we have designed and prototyped a language, semantics, matching mechanism, and matchmaking system that we collectively call *RedLine*. 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. Sections 3–5 describe the structure of the *RedLine* system, description language, and *RedLine* matchmaking process, respectively. In Section 6, we present applications used to evaluate our design. We conclude and outline our plans for future work in Section 7.

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 [29], LDAP [18], MDS [12], UDDI [21]). Such systems differ in various dimensions, such as their description syntax (e.g., MIBs [29], relations [11], LDAP objects [18]), query language (e.g., SQL [11], Xquery [28], LDAP query [18]), 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 resource. The consumer selection procedure may comprise a simple query or, alternatively, involve a procedural algorithm based, for example, on a performance model [2,6].

Although these information systems provide access control based on users' accounts/IDs, we argue that it are not enough for express and enforce resource provider policy, which restricts the availability of resource based on requesters' accounts/IDs, resource access time, and usage of this resource, etc. Furthermore, data access controls in these information systems are configured manually by the administrator who is usually not the same person as resource providers. 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 [20], 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 = [      owner = "chliu";
                 requirements = other.type=="machine" && other.cpuspeed > 500M;
                 rank = other.memsize ]
Resource = [     name="foo"; type="machine"; cpuspeed=800M;  memsize=512M;
                 requirements=member(other.owner, {"chliu", "lyang"})
                                  && DayTime() > '18:00'  ]
```

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

We can use *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 [23], LARKS [30], and the HP e-commerce matchmaker [13,25]. These systems use a range of different syntaxes (Java object, service profile, RDF, descriptive 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 [17] and W3C CC/PP [22], and/or to expressing and locating multiple resources with a particular relationship: the *resource co-selection* problem, as discussed by Dinda [9,24], Raman [26], and Czajkowski [12], among others.

We argue that no previous work solves these two problems completely. *Gang matching* [26] solves the resource co-selection problem by allowing a ClassAd to specify multiple resources, but not sets of resources defined by their aggregate characteristics; *set matching* [4] allows a ClassAd to specify resource sets, but not multiple resources of different types; neither can express and locate resources with a property value that is a feature set or range. 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 A New Approach to Matching

The essence of our approach is to treat matching as a constraint satisfaction problem and to apply constraint-solving technologies to implement resource search and selection functions.

3.1 Background on Constraints

Definition 1: A *constraint* C is of the form $c_1 \wedge \ldots \wedge c_n$ where n >= 0 and $c_1, \ldots, c_n$ are primitive constraints. The symbol $\wedge$ denotes *and*, so a constraint C holds whenever all of the primitive constraints $c_1, \ldots, c_n$ hold.

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.

The statement $(X=Y+2) \wedge X<3$ is an example of a constraint. It stipulates a relation that must hold between any values with which we choose to replace variable X and Y. This constraint is satisfiable because there exists an assignment X=2, Y= 0 that makes all primitive constraints hold.

Definition 3: A *constraint-solving algorithm* is an algorithm that, when given a constraint C, either finds an assignment to all variables such that this constraint holds, or finds this constraint is unsatisfiable.

Usually, we call a kind of constraint problems, in which the possible values of a variable are restricted to a finite set, as *constraint satisfaction problems*. Its definition is as follows.

Definition 4: A *constraint satisfaction problem*, or CSP, consists of a constraint C over variables $x_1, \ldots, x_n$ and a domain D that maps each variable x_i to a finite set of values, $D(x_i)$, that it is allowed to take. The CSP is understood to represent the constraint $C \wedge x_1 \in D(x_1) . \ldots . x_n \in D(x_1)$ [19], where $\wedge$ denotes "and" and $\in$ means "is an element of."

For example, the constraints, $C=\{x_1 > 1, x_1 + x_2 <4\}$, $D(x_1) = [1, 2, 3]$, $D(x_2) = [1, 2, 3]$, describe a CSP. The problem is to find a value for x_1 and x_2, subject to conditions that the value of x_1 must be bigger than 1 and the sum of x_1 and x_2 must be less than 4. The possible values for both x_1 and x_2 are 1, 2, and 3.

Constraints have been used to model many real-life problems, such as scheduling, routing, and timetabling, since these problems essentially involve choosing among a finite number of possibilities. Constraint-solving algorithms also have been developed in several research communities, including arc and node consistency techniques in the artificial intelligence community, bound propagation techniques in constraint programming community, and integer programming techniques in the operations research community [15,16,19].

3.2 Matching as Constraint Satisfaction

The matchmaking process is triggered by a resource request, in which the customer specifies resource requirements and interrelationships, such as "a CPU with 500 MHz CPU speed and a disk at least with 1000M free disk space, both located in the same domain." We must then select resources from a resource pool such that all request requirements and resource policies are satisfied.

We can formalize matchmaking (resource selection process) as a CSP by associating a variable with every requested resource. The domain of each variable is all available resources. Constraints on variables, which describe relations that must hold when choosing values for variables, express requirements concerning these resources and their access policies. In our example, we use two variables to express the required CPU and disk. Their domains are all CPU resources and disk resources, respectively. Constraints on the values of these two variables describe requirements. We can then use a constraint-solving algorithm to solve the problem.

We want a modeling language that allows for a declarative representation of both request(s) and resource description(s) and supports constraints on the record-like data types that are often required when describing a request for multiple related resources each with multiple properties. Thus, we do not adopt modeling languages such as OPL [14], Oz [3], and gprolog [8], as these only support constraints dealing with integer, real or string variables.

3.3 The RedLine System

As illustrated in Figure 2, the *RedLine* system has a layered architecture. The *RedLine language* defines the syntax and basic semantics for the specification of *descriptions*. A description is a set of constraints describing either a request or a resource. The syntax is fairly straightforward, allowing for the expression of a range of constraints on resource attributes.

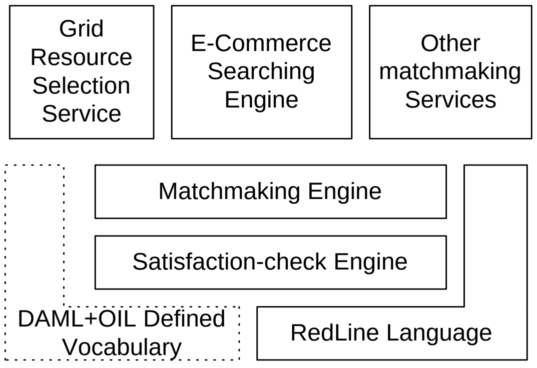
Grid Resource Selection Service
E-Commerce Searching Engine
Other matchmaking Services
Matchmaking Engine
Satisfaction-check Engine
DAML+OIL Defined Vocabulary
RedLine Language

Figure 2. Layered structure of the RedLine system

For added expressive power, the *RedLine* system allows for the specification of a *vocabulary* to be consulted when performing constraint checking. A vocabulary can use an ontology language such as DAML+OIL [5] and OWL [7] to define the semantics of words in a description: specifying, for example, that a string "Redhat" is a kind of Linux operating system.

The *Satisfaction-check engine* combines *RedLine language* statements and the semantic information defined in *vocabulary* to determine whether a constraint is satisfiable.

The *matchmaking engine* implements the logic used to match one request description with multiple resource descriptions. This process proceeds in two steps:

1. Map the resource selection problem at hand into a CSP problem that captures the required attributes of the request and existing resources.
2. Call the satisfaction-check engine to check whether a given assignment to variables causes conflicts.

These building blocks can be used to build a variety of different higher-level services, such as a resource selection service for Grid computing or a Web service discovery service. For example, in a Grid environment, *RedLine* matchmaker functions can be incorporated into an index node [12] that uses information service functions to maintain the information used for resource selection.

The semantic information defined in the (optional) vocabulary allows *RedLine* to perform semantic matches [5]. For example, a request for a resource with a Linux operating system will match a resource description that states its operating system as *Redhat* if Redhat is defined as a kind of Linux operating system in the vocabulary.

4 The RedLine Description Language

The design of the *RedLine* description language was informed by the following requirements:

- *Resource sets.* We want to be able to express requests that refer to multiple resources: for example, "a set of computers with total memory size bigger than 10G."
- *Ability to express resources property whose value is a feature set or range*.
- *Ability to describe requirements and preferences***.** An advertiser should be able to control what descriptions can match their description and the criteria to be used to select from among multiple matching descriptions.
- *Symmetric description of resource and resource request*. The same description syntax should be used to describe resources and requests. Thus, this language would allow both resource customers and resource owners to control what kind of descriptions can match their descriptions. Also, having the same syntax at both ends makes it easier for resource owners and customers to query and understand descriptions of their counterparts.

4.1 RedLine Grammar

We describe in turn the *RedLine* type system, set-related functions, attributes, constraints, and descriptions.

4.1.1 Types

In addition to the usual base types (real, integer, string, Boolean), *RedLine* defines the following collective types:

- *Description*, a finite set of statements about properties of an entity, used to describe one or a kind of resources. See Section 4.1.4 for details.
- *Set*, an unordered sequence of one or more values or expressions, within which a given value can appear only once (i.e., an attempt to place a value into a set more than once is ignored). A set is constructed as follows:

  [expr, expr, …, expr]: a set consisting of evaluation results of *expr*s.
- *Enumeration*, used to define a variable whose value can only be chosen from a finite set of values. Enumeration variables may only contain values that are defined in the enumeration. For example, in the statement:

  os=ENUM["linux", "windows", "unix"]

  Variable *os* may be assigned only the values **"***linux", "windows" or "unix"*.
- *Dictionary*, a set of key-value pairs. For example, the following assigns a dictionary value to the identifier *bandwidth*:

bandwidth= DICTIONARY[{“trapezius”, 10}, {“vatos”, 10}]

We can access a value by reference to its key, as in

x = bandwidth(“trapezius”)   // here x is assigned to 10

Typing in the *RedLine* language is dynamic: that is, the types of variables are defined by usage, not declaration.

4.1.2 Attributes and Constraints

RedLine uses attributes to describe resource properties. In matchmaking, it is not always possible or preferable to describe an attribute by a particular value. Unlike other languages that use attribute-value pairs to describe entities, *RedLine* uses constraints that describe a relation that must hold when choosing values for variables:

Constraint ::= variable ‘=‘ expr
| *logicexpr*
| *predicate*

Expr is an arithmetic expression with operators as “+”, “-“, “*”, “/” and “^”. The operands can be of type integer or real, for example,

a= 2: value of *a* is equal to 2. An assigned variable is maximally constrained: no further non-redundant constraints can be imposed on the variable, without introducing an inconsistency.

a=b+c: value of *a* is equal to sum of value of *b* and *c*; value of *b* is equal to difference of value of *a* and *c*; value of *c* is equal to difference of value of *a* and *b*.

LogicExpr is a logical expression with operators as “>”, “<”, “>=”, “<=”, “==”, “&&”, “||”, and “!”. The operands of logic expression can be of type integer, real, Boolean, and string. For example:

a>100: value of *a* is bigger than 100.

a > b+c: value of *a* is bigger than the sum of *b* and *c*.

Predicate is a system-defined constraint. We define the following predicates.

- *Minimize(<expr>)*: Multiple values for variables are possible; choose the one minimizing the value of expression <*expr*>.
- *Maximize(<expr>)*: When there are multiple possible value for variables, choose the one maximizing the value of expression <*expr*>.
- *Forall x in <set>*: All elements in <*set*> are subjected to constraints related to *x*. For example, *x.cpuspeed > 100* means all elements in <*set*> have an attribute named *cpuspeed* with value bigger than 100.
- *Forany x in <set>*: One or more elements in <*set*> are subjected to constraints related to x. For example, *x.cpuspeed > 100* means there is at least one element in <*set*> having an attribute *cpuspeed* with value bigger than 100.
- *Required(<set of attribute>)*: All attributes listed in <*set of attributes*> must appear in the other description involving in a match. See Section 5.1 for details.

Users can use *Minimize* and *Maximize* to describe their preferences to resources if multiple choices are available. *Forall* and *Forany* are used to describe constraints on elements in a set. *Required* is used to specify the necessary information for a successful match in resources (see 5.1 for details).

4.1.3 Set Functions

In order to describe characteristics of resource sets, *RedLine* defines the following set-related functions:

- *Count(<set>)*: returns the number of elements in a set <*set*>.
- *Max(<set>)*: returns the maximum value of a integer/real set <*set*>, **error** otherwise.
- *Min(<set>)*: returns the minimum value of a integer/real set <*set*>, **error** otherwise.
- *Sum(<set>)*: returns the average value, expressed as a real value, of a integer/real set <*set*>, **error** otherwise.
- *InSet(<set>, <value>)*: returns **true** if <*value*> is an element of <*set*>, **false** otherwise.
- *Set_Intersection(<seta>, <setb>)*: returns the intersection of <*seta*> and <*setb*>.
- *Set_Union(<seta>, <setb>)*: returns the union of <*seta*> and <*setb*>.
- *Set_Difference(<seta>, <setb>)*: returns the set consisting of elements in <*seta*> and not in <*setb*>.
- *Set_S_Difference(<seta>, <setb>)*: returns the set consisting of elements that are in <*seta*> and not in <*setb*> or in <*setb*> and not in <*seta*>.

4.1.4 Description Structure

A description is a set of statements in which each statement is expressed by a constraint. The syntax of construction of a description is as follows:

description=[constraints1; constraints2; …; constraints3]

Description is also a data type. A description value can be assigned to a variable as follows:

A=[cpuspeed=1; memsize=2]

A new operator *ISA* is used to specify the constraints to a *description* type or *description set* type variable as '*<variable> ISA <description>*' and '*<variable> ISA SET<description>*'. The meaning is illustrated by the following two examples.

r ISA [os="linux"; memsize>1G] means that the value of *r* is a description with attribute *os* equal to *"linux"* and attribute *memsize* bigger than 1G.

t ISA SET[os="linux"; memsize > 1G] means that the value of *t* is an description set whose elements have attribute *os* equal to "*linux*" and attribute *memsize* bigger than 1G.

These two kinds of constraint on the *description* type of variables, *assignment* and *ISA*, constrain variables differently. An assignment to a variable constrains the variable maximally; any other nonredundant assignment then results in an inconsistency. In contrast, an ISA-constrainted variable can be refined by more constraints. For example, *A=[cpuspeed=1; memsize=2; os="linux"]* will conflict with constraint *A=[cpuspeed=1; memsize=2]*; however, a constraint *r ISA [os="linux"; memsize>1G]* can be refined by constraint *r = [os="linux"; memsize= 2G; cpuspeed=500]* without inconsistency.

We can refer to attribute values in a description and a description set by the "." operator. Thus, in the preceding example, *r.os* evaluates to *"linux*" and *t.memsize* to a list comprising the values of the attribute *memsize* for every element in set *t*.

4.2 Examples of RedLine Descriptions

Every *RedLine* description can be interpreted as a resource advertisement as well as a request. A description's role may be determined by how the description is sent to the matchmaker: descriptions submitted through a resource advertising interface are *resource* descriptions, and descriptions submitted through request advertising interface are *request* descriptions. In order for the description to be understandable by all users, all advertisers need to use the same terminology.

Request

```
[user="globus-user";
 group="dsl-uc";
 computation ISA SET[type="computation"];
 storage ISA [ type="storage"; space > 100];
 Forall x in computation;
 x.cpuspeed > 150;
 x.bandwidth[storage.hn] > 30;
 x.accesstime > 18;
 Sum(computation.memory) > 300;
 storage.space > 80;
 storage.accesstime > 18 ]
```

Resources

```
R1= [type="computation"; hn="c1.uchicago.edu";
       cpuspeed=200;
       bandwidth= DICTIONARY[
                   {"s1.uchicago.edu", 20},
                   {"s2.uchicago.edu", 40} ];
       accesstime > 17 ]
R2= [type="computation"; hn="c2.uchicago.edu";
       cpuspeed=200;
       bandwidth=DICTIONARY[
                   {"s1.uchicago.edu", 20},
                   {"s2.uchicago.edu", 40} ];
       accesstime > 17]
R3= [type = "storage"; hn="s1.uchicago.edu"; space=100]
R4= [type = "storage"; hn="s2.uchicago.edu"; space=200]
```

Figure 3. Examples of RedLine description

Figure 3 shows a *RedLine* description specifying a resource request that requires a set of computation resources with CPU speed faster than 150 MHz and total memory size bigger than 300 Mbytes, a storage resource with space bigger than 80 G, and a network connection between every computation resource and the storage resource that is faster than 30 Kbytes per second. The access time to these resources is after 6:00PM. ClassAds cannot describe multiple resources in this way. Furthermore, not only can *RedLine* describe requests for multiple resources of different types, it can also describe resource set with aggregate characteristics.

Figure 3 also shows four simple examples of resource descriptions: two computers (R1 and R2) and two storage systems (R3 and R4). These examples illustrate how *RedLine* expresses both access policy (*accesstime)* and properties (such as *cpuspeed*) in the same way, thus allowing a user to query both policy and properties. Please refer to examples in Section 6 for details.

5 Matchmaking in RedLine

5.1 Definition of Match

A *RedLine* description is a self-consistent collection of constraints over named properties of an entity [25]. A description D_1 *matches* a description D_2 if constraints $D_1 \wedge D_2$ is satisfiable. This definition allows a match to proceed if D_1 specifies constraint(s) on an attribute A, and D_2 does not: the match fails only if $D_1 \wedge D_2$ contains constraint(s) on A that turn out to be mutually inconsistent. This approach is consistent with the observation that we often want to allow matches between descriptions with different level of generality and complexity. For example, the simple request description *[type="computation"]* will match all computational resources descriptions that contain a constraint *type="computation."*

In some situations, we may not want a match to succeed simply because the other participant(s) do not specify constraints on an attribute. Thus, *RedLine* provides a constraint *Required(<attribute-set>)* that requires all attributes listed in *<attribute-set>* to appear in the matching description. Users can use this constraint to limit the matched result to descriptions with particular information. For example, a request *[type="computation"; Required(os)]* won't match the computational resource descriptions R1 and R2 in Figure 3 because there is no property *os* in these two descriptions.

RedLine also defines multilateral match: Descriptions D_1, D_2, … , D_n *match* a description R if D_1, D_2, …, D_n is an assignment to variables with description or description set type in description R and R is still satisfiable after replacing these variables with their values. For example, in Figure 3, resource descriptions R1, R2, and R4 match the request because assigning R4 to attribute *storage* and set [R1, R2] to attribute *computation* satisfies all the constraints in these descriptions.

5.2 Matchmaking Process

In Section 3, we presented the syntactic basis for *RedLine* matchmaking. We now consider the problem of efficiently identifying matched resource descriptions for a request. We focus here on multilateral matchmaking and treat bilateral match as a special case.

Multilateral matchmaking is triggered by a request description that describes multiple resources and their relationship, such as Figure 3. In a request, resources are associated with variables by constraints '<*variable*> **ISA** <*description*>' or '<*variable*> **ISA** SET<*description*>.' In order to distinguish these variables from those used to describe resource attributes, we call them as *resource variables*. As defined in Section 5.1, matchmaking seeks to find values for these resource variables.

Matchmaking proceeds in two steps. First, the matchmaker decides the domain of these variables. For variables described by '<*variable*> **ISA** <*description*>', the matchmaker algorithm treats resources that match description <*description*> as the value domain of <*variable*>. For variables described by '<*variable*> **ISA** SET<*description*>', the matchmaker uses all resources that match description <*description*> to construct candidate sets as the value domain of <*variable*>. Candidate set construction is a complex computation. Assume R is a set including all resources matching <*description*>, candidate sets for <*variable*> are subsets of R. Because the number of subsets of R is exponential to its cardinality, we use a heuristic method suggested by Dail {Dail, 2002 #41} and Liu {Chuang Liu, 2002 #39} to create candidate sets. The basic idea is to rank all resources firstly based on some criteria, then construct a candidate set V_i (i=1 to number of resources in R) including the first i best resources in R. So the cardinality of value domain is linear to the number of resources.

In the second step, the matchmaker checks whether there exists a conflict-free assignment to these resource-associated variables. As mentioned in Section 3, this is a constraint satisfaction problem (or constraint optimization problem if constraint Maximize() or Minimize() is specified). CSP research community has developed a lot of efficient algorithms to solve the combinatorial search problems. One of the benefits to model matchmaking into a CSP is to utilize these existing algorithms to implement the matchmaking process. In our *Redline* prototype, we have implemented the matchmaking process as follows.

1. Use a node consistency algorithm [19] to reduce the domain of every variable. This algorithm uses constraints involving only one variable to reduce the domain of variables. For example, in Figure 3, constraint *storage.space > 80* is used to remove all storage resources with space less than 80 from value domain of variable *storage*.
2. Use a backtracking method [19] to solve the problem. Here, algorithm performance is critically dependent on the order in which both the variable and its value are picked. We implemented the first-failing algorithm that starts backtracking from the variable with the smallest domain, and for a variable we chose a value from its domain randomly.

Assume a request with K resource variables and a resource pool including N resources, the complexity of this matchmaking process is $O(N^k)$. So for a given request, the algorithm complexity is polynomial to the number of resources. For scenario where N is huge and complete algorithm is not required, other heuristic and stochastic algorithm for CSP problem is available, such as Hill-Climbing, Min-Coflict and Tabu-Search {BARTÁK, 1998 #131}, etc. Although we have not implement these algorithms, we argue that by modeling matchmaking into a CSP problem, we relieve ourselves from algorithm issues.

Preliminary experiments with this prototype show that performance is at least comparable to that of the Condor ClassAd matchmaking system on several large-scale resource selection problems. However we do not yet have a detailed understanding of *RedLine* performance.

6 Applications

No standard set of challenge problems for matchmaking has been defined. Thus, a comprehensive evaluation of the usability of the *RedLine* language will require extensive practical experimentation in multiple different application domains—experimentation that we have not yet undertaken. Here, however, we use three examples to demonstrate the range of matching problems that can be expressed in this language.

Our first example shown in Figure 4 illustrates the ability of the *RedLine* language to express and locate resource whose properties value is a feature set and range:

```
resource = [
   type="displayDevice";
   resolution = ENUM [ [pix-x<=640; pix-y<=480; color<= 24],
                       [pix-x<=800; pix-y<=600; color<= 16],
                       [pix-x<=1024; pix-y<768; color<=8] ] ];
request = [
   rs ISA [type="displayDevice"]; rs.resolution.pix-x =750;
   rs.resolution.pix-y=500; rs.resolution.color=4 ]
```

Figure 4. Display Device Example

Description *resource* describes a screen capable of displaying 640*480 pixels and 16 million colors (24 bits per pixel), 800*600 pixels and 64 thousand colors (16 bits per pixel), or 1024*768 pixels and 256 colors (8bits per pixel). Description request describes a request for a display device that can show 750*500 pixel image using 16 colors. This request will match this screen.

Figure 5 illustrates the use of ranking criteria to instruct the matchmaking process to select the “best” resource. We show three resource descriptions and a request description. The request expresses the user’s requirement to find a site that has the maximum number of a set of specified input data. It also requires, as an additional constraint, that the site have a minimum amount of free space. This code might be used, for example, within a Data Grid system to decide where to send a task. Note how easily the criteria used to select the destination site can be changed.

```
site1= [ type="site"; name="site1"; space = 100; data=["a", "b", "c"] ]
site2= [ type="site"; name="site2"; space = 200; data=["c", "d"] ]
site3= [ type="site"; name="site3"; space =300; data=["b" ]]
request = [
   site ISA [ type="site"];
   site.space > 10;
   wantedData =["a", "b"];
   availableData= Count(Set_Intersection( wantedData, site.data));
   Maximize(availableData) ]
```

Figure 5. Data Grid example

Figure 6 illustrates *RedLine*'s support for query function. Description *RS* describes a resource with CPU speed 500 MHz and an access policy that states that it is available only to users "*globus*" and "*dsl-uc*" after 6:00 PM. Description *Q1* is a query for a resource available after 8:00 PM. Description *Q2* is a query for a resource that has a CPU speed faster than 400 MHz. Both *Q1* and *Q2* will find *RS* because description *Q1* and *Q2* match *RS*. Note that both access policy (*accesstime* in *Q1*) and resource property (*cpuspeed* in *Q2*) can be used as criteria to query resources.

```
RS= [ cpuspeed = 500; accesstime > 18;
        permittedUser= ENUM[ "globus", "dsl-uc"]]
Q1= [ accesstime > 20]
Q2= [ cpuspeed > 400 ]
```

Figure 6: Query Examples

7 Summary and Future Work

Resource selection in Grid environments usually involves multiple resources with diverse ownership and policies. We have designed and implemented a description language, *RedLine*, for expressing constraints associated with resource consumers (requests) and resource providers. We have also implemented a matchmaking process that uses constraint-solving techniques to solve the combinatorial satisfaction problems that arise when resolving constraints. The resulting system has significantly enhanced expressiveness compared with previous approaches, being able to deal with requests that involve multiple resources and that express constraints on policies as well as properties. *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; and improving our constraint-solving algorithm to address matchmaking performance requirements identified in realistic application settings.

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 Alain Roy for comments on a draft of this paper.

Reference

[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] Christian Schulte and Smolka, G., Finite Domain Constraint Programming in Oz. 2002.

[4] Chuang Liu, Lingyun Yang, Ian Foster 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.

[5] Connolly, D., Harmelen, F.v., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F. and Stein, L.A., DAML+OIL (March 2001) Reference Description. W3C, 2001.

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

[7] Dean, M., Connolly, D., Harmelen, F.v., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F. and Stein, L.A., Web Ontology Language (OWL) Reference Version 1.0. W3C, 2002.

[8] Diaz, D., GNU PROLOG. 2002.

[9] Dinda, P. and Plale, B., A Unified Relational Approach to Grid Information Services. Grid Forum Informational Draft GWD-GIS-012-1, 2001.

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

[11] Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database systems: the complete book*, Prentice Hall, Upper Saddle River, NJ, 2002, xxvii, 1119 pp.

[12] Globus, MDS document. www.globus.org/mds.

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

[14] Hentenryck, P.V., *The OPL Optimization Programming Language*, The MIT Press, Cambridge, Massachusetts, 1999.

[15] Hentenryck, P.V., Michel, L., Perron, L. and Regin, J.-C., Constraint Programming in OPL. *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, Paris, France, 1999.

[16] Henz, M. and Müller, T., An Overview of Finite Domain Constraint Programming. *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore, 2000.

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

[18] 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.

[19] Kim Marriott and Peter, J.S., *Programming with Constraints: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie and Siméon, J., XQuery 1.0: An XML Query Language. W3C, 2002.

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

[30] 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.