

# A Constraint Language Approach to Grid Resource Selection

Chuang Liu<sup>1</sup> Ian Foster<sup>1,2</sup>

<sup>1</sup>Department of Computer Science, University of Chicago, Chicago, IL 60637

<sup>2</sup>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 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, *RedLine*, based on this approach and have conducted preliminary experiments that suggest that it can be implemented at least as efficiently as ClassAds. We have also successfully applied *RedLine* to some challenging matching problems from several application domains.

## 1 Introduction

The development of Internet and Grid technology [9] 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 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 [18,21,22], 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 a generalized constraint satisfaction problem. We describe a

new matching system within which we can do the following, none of which are supported within ClassAds.

- *Describe resources with different levels of generality and complexity.* ClassAds perform only exact matches on properties. Yet different providers and consumers may use descriptions with varying levels of generality and complexity. For example, while an individual may specify: “1995 Corolla, first owner, 60,000 miles, want \$10,000,” a dealer may prefer more general descriptions: “Cars produced by Toyota and Honda, with price from \$15,000 to \$25,000.” We support descriptions with different levels of generality and complexity.
- *Match advertisements based on policy as well as properties.* ClassAd encodes 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 conducted experiments designed to characterize the performance characteristics of our prototype, and we have applied *RedLine* to some challenging matching problems from several application domains. Our preliminary results indicate performance similar to that of the Condor matchmaker and significantly better expressivity.

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 the benchmark problems and 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 service descriptions (SNMP [24], LDAP [15], MDS [11], UDDI [19]). Such systems differ in various dimensions, such as their description syntax (e.g., MIBs [24], relations [10], LDAP objects [15]), query language (e.g., SQL [10], Xquery [23], LDAP query [15]), and the techniques used to publish and aggregate service descriptions (e.g., soft state vs. stateful servers, complete descriptions vs. Bloom filters). However, these systems all have in common that the service 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 service. The consumer selection procedure may comprise a simple query or, alternatively, involve a procedural algorithm based, for example, on a performance model [2,6]. Such strategies require that service provider policy be enforced only *after* candidate services are selected by the consumer, which can result in wasted effort.

**Symmetric evaluation.** Symmetric evaluation was pioneered by the Condor *matchmaker* system [18], 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.**

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.

Condor matchmaking can describe only binary matches. *Gang matching* [21] overcomes this limitation 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. More significant, the asymmetric treatment of properties and requirements (properties are defined by expressions, while requirements are bundled up in the *requirements* statement) makes it difficult to query requirements information. For example, given a resource description such as *Resource* in Figure 1, we might want to express the request “Find all machines accessible after 7:00 PM.” But Condor matchmaking does not allow matching on information in a requirements statement: the author of the *ClassAd resource* would have to duplicate that information in new properties. Putting all requirements in a statement also make it difficult to identify unfulfilled constraints for two descriptions that fulfill partly their counterparts’ requirements.

**Knowledge representation.** SNMP MIBs, LDAP object classes, and *ClassAds* are all simple knowledge representation formalisms. More sophisticated formalisms, such as semantic web technology [17], have been developed that allows for the representation and organization of more complex information about objects. Semantic web is an extension of current web technology, in which information is given well-defined meaning. Semantic web technology can be used to described data, service, resource, etc. It enables the query and matching of resources based on their capability rather than their syntactical expression [1]. Although semantic web technology allows more precise resource selection, it doesn’t address the resource co-selection problem.

### 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

The constraint satisfaction problem is defined as follows.

**Definition 1:** A *constraint satisfaction problem*, or CSP, consists of a constraint  $C$  over variables  $x_1, \dots, x_n$  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  $C \wedge x_1 \in D(x_1) \dots x_n \in D(x_n)$  [16], 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.

**Definition 2:** A *constraint-solving algorithm* is an algorithm that, when given a valid CSP, either finds an assignment to all variables such that no constraint is violated, or fails.

Applying a constraint-solving algorithm to the previous example will yield the result that  $x_1$  is equal to 2 and  $x_2$  is equal to 1.

Constraint satisfaction problems 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 [13,14,16].

#### 3.2 Matching as Constraint Satisfaction

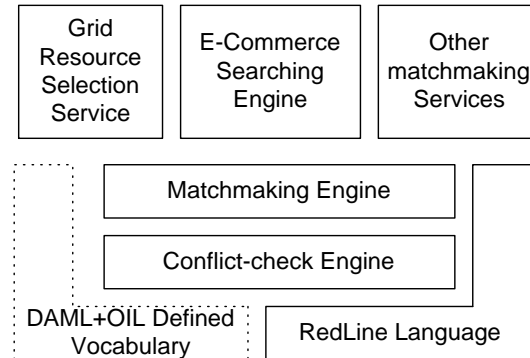
The matchmaking process is triggered by a resource request, in which the customer specifies resource requirements and interrelationships, such as “a CPU with 500 MHz CPU speed and a disk at least with 1000M free disk space, both located in the same domain.” We must then select resources from a resource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

We can formalize matching 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 support constraints on the structured data types that are often required when describing complex resources. Thus, we do not adopt modeling languages such as OPL [12], Oz [3], and gprolog [8], as these require that users write programs to solve problems and furthermore only support constraints dealing with integer or ground term valued 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.



**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 *conflict-check engine* combines *Redline language* statements and the semantic information defined in *vocabulary* to determine whether a set of constraints is consistent.

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 conflict-check engine to check whether a given assignment to all 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 search engine for a Web service.

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 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.
- *Support for query of descriptions.* One should be able to query descriptions based on some criteria about their properties and requirements.

## 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* includes two predefined values:

- **undefined**, generated when referring to an attribute whose value is not decided; and
- **error**, generated when there is a type error in expression evaluation.

It also defines the following collective types:

- *Description*, a finite set of statements about properties of an entity, used to describe one or a kind of entity. 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 *exprs*.

- *List*, represented as *{expr, expr, ... expr}*, an ordered sequence of zero or more expressions, within which any given item can appear any number of times. List elements may be accessed by *<list>[i]*.
- *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 Set Functions

*RedLine* defines the following set-related functions:

- *Count*( $\langle set \rangle$ ): returns the number of elements in a set  $\langle set \rangle$ .
- *Max*( $\langle set \rangle$ ): returns the maximum value of a integer/real set  $\langle set \rangle$ , **error** otherwise.
- *Min*( $\langle set \rangle$ ): returns the minimum value of a integer/real set  $\langle set \rangle$ , **error** otherwise.
- *Sum*( $\langle set \rangle$ ): returns the average value of a integer/real set  $\langle set \rangle$ , **error** otherwise.
- *InSet*( $\langle set \rangle$ ,  $\langle value \rangle$ ): returns **true** if  $\langle value \rangle$  is an element of  $\langle set \rangle$ , **false** otherwise.
- *Set\_Intersection*( $\langle seta \rangle$ ,  $\langle setb \rangle$ ): returns the intersection of  $\langle seta \rangle$  and  $\langle setb \rangle$ .
- *Set\_Union*( $\langle seta \rangle$ ,  $\langle setb \rangle$ ): returns the union of  $\langle seta \rangle$  and  $\langle setb \rangle$ .
- *Set\_Difference*( $\langle seta \rangle$ ,  $\langle setb \rangle$ ): returns the set consisting of elements in  $\langle seta \rangle$  and not in  $\langle setb \rangle$ .
- *Set\_S\_Difference*( $\langle seta \rangle$ ,  $\langle setb \rangle$ ): returns the set consisting of elements that are in  $\langle seta \rangle$  and not in  $\langle setb \rangle$  or in  $\langle setb \rangle$  and not in  $\langle seta \rangle$ .

### 4.1.3 Attributes and Constraints

*RedLine* uses attributes to describe entity 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 have 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  $b$ .

$k = \text{ENUM}[\text{“Chicago”}, \text{“New York”}]$ :  $k$  is a string with value “Chicago” or “New York.”

*LogicExpr* is a logical expression with operators as “>”, “<”, “>=”, “<=”, “==”, “&&”, “||”, and “!”. The operands of logic expression can be as integer, real, Boolean, and string type. 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*( $\langle expr \rangle$ ): Multiple values for variables are possible; choose the one minimizing the value of expression  $\langle expr \rangle$ .
- *Maximize*( $\langle expr \rangle$ ): When there are multiple possible value for variables, choose the one maximizing the value of expression  $\langle expr \rangle$ .
- *Forall*  $x$  in  $\langle set \rangle$ : All elements in  $\langle set \rangle$  are subjected to constraints related to  $x$ . For example,  $x.cpuspeed > 100$  means all elements in  $\langle set \rangle$  have an attribute named *cpuspeed* with value bigger than 100.
- *Forany*  $x$  in  $\langle set \rangle$ : One or more elements in  $\langle set \rangle$  are subjected to constraints related to  $x$ . For example,  $x.cpuspeed > 100$  means there is at least one element in  $\langle set \rangle$  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.

#### 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*-constrained 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 description in *RedLine* 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.

```
[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 ]
```

**Figure 3. A RedLine request 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 M, 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.

```

R1= [type="computation"; hn="ucsd1"; cpuspeed=200;
      bandwidth=DICTIONARY[{"s1", 20}, {"s2", 40}];
      accesstime > 17 ]
R2= [type="computation"; hn="ucsd2"; cpuspeed=200;
      bandwidth=DICTIONARY[{"s1", 20}, {"s2", 40}];
      accesstime > 17]
R3= [type = "storage"; hn="s1"; space=100]
R4= [type = "storage"; hn="s2"; space=200]

```

**Figure 4. Four examples of RedLine resource descriptions**

Figure 4 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. See Section 6.2 for details.

## 5 Matching and Matchmaking in RedLine

We describe two *RedLine* functions: matching and matchmaking.

### 5.1 Matching

A *RedLine* description is a self-consistent collection of constraints over named properties of an entity [20]. A description  $D_1$  *matches* a description  $D_2$  if there is no conflict between the constraints  $D_1 \cup D_2$ . 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  $D_1 \cup D_2$  each 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 4 because there is no property *os* in these two descriptions.

*RedLine* also defines multilateral match: Descriptions  $D_1, D_2, \dots, D_n$  *match* a description  $R$  if  $D_1, D_2, \dots, D_n$  is an assignment to all variables with description or description set type in description  $R$  and causes no conflict. For example, R1, R2, and R4 in Figure 4 match the request in Figure 3 because assigning R4 to attribute *storage* and set [R1, R2] to attribute *computation* satisfies all the constraints in these descriptions.

## 5.2 Matchmaking

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 entities and their relationship, such as Figure 3. As defined in Section 5.1, matchmaking seeks to find values for variables with description or description set type in the request description. Matchmaking proceeds in two steps. First, the matchmaker decides the domain of all variables with description or description set type whose value is undecided. In *Redline*, these variables are described by constraint ‘<variable> **ISA** <description>’ and ‘<variable> **ISA SET**<description>’. 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>.

In the second step, the matchmaker checks whether there exists a conflict-free assignment to all variables with description or description type. As mentioned in Section 3, this is a constraint satisfaction problem (or constraint optimization problem if constraint Maximize() or Minimize() is specified). Because solving CSPs is NP-hard [16], it is unlikely that there is a polynomial algorithm to find a solution for an arbitrary CSP, although several efficient algorithms are widely used to model and solve CSPs in constraint programming.

We implement the two steps of the matchmaking process as follows.

1. Use a node consistency algorithm [16] 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 [16] 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.

## 6 Experimental Evaluation

We used two microbenchmarks to evaluate our *RedLine* implementation. We also illustrate the system’s capability by using *RedLine* to solve three real-world applications.

### 6.1 Microbenchmarks

In view of the possibly huge number of resources and requests, a matchmaking system must be efficient and scalable. In this section, we test these features on two microbenchmarks.

Our first experiment compares the efficiency of the *RedLine* matchmaker with that used in the well-known Condor ClassAds system. We choose a simple bilateral match problem in which the request specifies the identity of the requestor and a requirement that a resource have memory greater than 100 Mbytes, and the resource description states a resource with 200 MBytes memory size and an access policy that allows access to only two particular users. As shown in Figure 5, this problem can be expressed in both *RedLine* and ClassAds.

| Type     | Resource and Request Description  | Time/match |
|----------|---|------------|
| RedLine  | Resource = [ mem= 200; user=ENUM["chliu", "lyang"] ]<br>Request = [ mem > 100; user="chliu"]  | 1.4 ms     |
| ClassAds | Resource = [mem=200; requirements=other.user=="chliu"    other.user=="lyang"]<br>Request = [user="chliu"; requirements=other.mem > 100] | 3.7 ms     |

**Figure 5. Performance of bilateral match**

Our experiments were conducted on a Linux system (550 MHz AMD-K6 CPU and 128 Mbytes memory) using our *RedLine* prototype and the Condor ClassAds library (version 0.9.2). In each case, we performed 10,000 repetitions of the simple match. Per match costs were 1.4 ms for *RedLine* and 3.7 ms for ClassAds. In this bilateral matching problem, then, *RedLine* is faster than ClassAds.

The second experiment illustrates the scalability of our matchmaker implementation when handling multilateral matching. This benchmark problem is motivated primarily by a real-world license management problem [21]. The scenario consists of a number of jobs each of which requires a machine and a license to run the application. Licenses have different types, and each type of license is valid only on some subset of machines. Thus the workstation and license resources required by each job are interdependent.

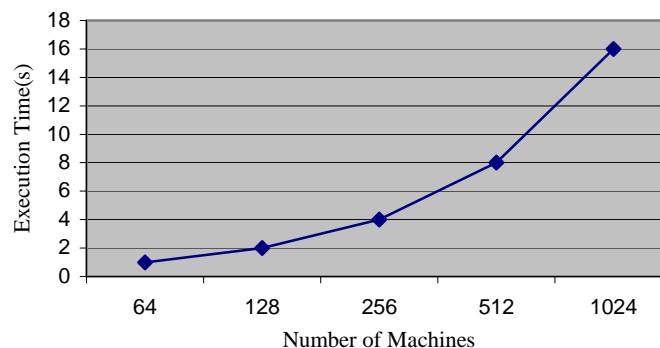
We simulated the machine pool based on data collected from a Condor resource pool by Raman that is described in [21]. Each machine description specifies the operating system (attribute *os*) and memory size (*mem*). Each machine is also assigned a unique integer key, hosted, in  $[0, n)$ , where  $n$  is the number of machines being modeled, for example,

[ hostID = 1; os="LINUX"; mem=128]

Licenses have four different types (type 0, 1, 2, 3); type  $i$  is valid for a machine whose *hostID* is equal to  $(i \bmod 4)$ . For simplicity, we assume that the number of licenses is unlimited.

The job request describes the requirement for a license and a machine. The requirement for the memory of machine is a random value between 0 and the maximum memory size of machine in the resource pool. The operating system requirement is generated using the same probability distribution as that used to simulate the resource pool. The number of requests is 64.

The experimental results are shown in Figure 6, in which the x axis is the number of machines in the simulated machine pool shown in log scale and the y axis is the time required to match all 64 requests. The results show that the matchmaker in *RedLine* system can handle these requests in reasonable time and that matchmaking time is linear with the number of machines.



**Figure 6. Performance of multilateral match**

## 6.2 Applications

In the absence of a well-defined set of target problems for matchmaking, 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, a description from a car dealer, illustrates the descriptive ability of the *RedLine* language to express properties that cannot be described by an attribute/value pair:

```
[ item="car"; brand="BMW"; color=ENUM[ "grey", "white", "red"]; price > 20000; price < 28000 ]
```

This description states that the dealer is selling BMWs with price from \$20,000 to \$28,000 and with colors grey, white, or red.

Figure 7 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 7. Data Grid example**

Figure 8 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 8: 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. Initial performance studies are encouraging.

More work is required to evaluate the effectiveness of the *RedLine* system in a wider range of applications, and to complete 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.

## Reference

- [1] Ankolekar, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D.L., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T., Sycara, K. and Zeng, H., DAML-S: Semantic Markup for Web Services. *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Stanford University, California, USA, 2001.
- [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] Foster, I. and Kesselman, C., *The Grid: Blueprint for A New Computing Infrastructure*, Morgan Kaufmann Publishers, San Francisco, 1999, xxiv, 677 pp.
- [10] Garcia-Molina, H., Ullman, J.D. and Widom, J., *Database systems: the complete book*, Prentice Hall, Upper Saddle River, NJ, 2002, xxvii, 1119 pp.
- [11] Globus, MDS document. [www.globus.org/mds](http://www.globus.org/mds).
- [12] Hentenryck, P.V., *The OPL Optimization Programming Language*, The MIT Press, Cambridge, Massachusetts, 1999.
- [13] 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.

- [14] 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.
- [15] 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.
- [16] Kim Marriott and Peter, J.S., *Programming with Constraints: An Introduction*, The MIT Press, Cambridge, Massachusetts, 1998.
- [17] Koivunen, M.-R. and Miller, E., W3C Semantic Web Activity. *Semantic Web Kick-off Seminar in Finland*, Helsinki, Finland, 2001.
- [18] 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.
- [19] Newcomer, E., *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, Boston, 2002, xxviii, 332 pp.
- [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] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie and Siméon, J., XQuery 1.0: An XML Query Language. W3C, 2002.
- [24] Stallings, W., *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, 3rd edn., Addison-Wesley, Reading, Mass., 1999, xv, 619 pp.