DataGrid

JOB DESCRIPTION LANGUAGE HOWTO

Document identifier:	**DataGrid-01-TEN-0102-0_2**
Date:	**17/12/2001**
Work package:	**WP1**
Partner:	**Datamat SpA**
Document status	**DRAFT**
Deliverable identifier:	

Abstract: This note provides a description of the DataGrid Job Description Language

Delivery Slip

	Name	Partner	Date	Signature
From	Fabrizio Pacini	Datamat SpA	17/12/2001	
Verified by	Stefano Beco	Datamat SpA	17/12/2001	
Approved by				

Document Log

Issue	Date	Comment	Author
0_0	28/05/2001	First draft	Fabrizio Pacini
0_1	13/09/2001	Annex on JDL attributes updated	Fabrizio Pacini
0_2	17/12/2001	Document updated according to comments received from Applications WPs.	Fabrizio Pacini

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files
Word 97	DataGrid-01-TEN-0102-0_2-Document
Acrobat Exchange 4.0	DataGrid-01-TEN-0102-0_2-Document.pdf

CONTENT

1. INTRODUCTION

The growing emergence of large scale distributed computing environments such as *computational grids*, presents new challenges to resource management, which cannot be met by conventional systems that employ relatively static resource models and centralised allocators.

A principal consideration of resource management systems is the efficient assignment of resources to customers. The problem of making such efficient assignments is referred to as the *resource allocation* problem and it is commonly formulated in the context of a *scheduling model* that includes a *system model*. A *system model* is an abstraction of the underlying resources, to describe the availability, performance characteristics and allocation policies of the resources being managed.

In a distributively owned environment, the owner of a resource has the right to define its usage policy, which may be very sophisticated. For example, the policy may state that a job can run on a workstation only if it belongs to a particular research group, or if it is run between a well-determined time period of the day. Distributed ownership together with heterogeneity, resource failure and evolution make it impossible to formulate a monolithic system model, there is therefore a need for a resource management paradigm that does not require such a model and that can operate in an environment where resource owners and customers dynamically define their own models.

A fundamental notion for workload management in any such distributed and heterogeneous environment is entities (i.e. servers and customers) description, which is accomplished with the use of a description language. In the following of this document we describe in detail the design goals, structure and semantics of a Job Description Language that can be used as the language substrate of distributed frameworks, the *Classad* language.

The *classified advertisement* (*classad*) language is a symmetric description language (both servers and customers use the same language to describe their respective characteristics, constraints and preferences) whose central construct is the *classad,* a record-like structure composed of a finite number of distinct *attribute names* mapped to expressions. A *classad* is a highly flexible and extensible data model that can be used to represent arbitrary services and constraints on their allocation.

Main novel aspects of this framework can be summarised by the following three points that will be detailed in the next sections:

- Classads use a semi-structured data model, so no specific schema is required by the resource management system, allowing it to work naturally in a heterogeneous environment
- The classad language folds the query language into the data model. Constraints (i.e., queries) may be expressed as attributes of the classad.
- Classads are first-class objects in the model. They can be arbitrarily nested, leading to a natural language for expressing resource aggregates or co-allocation requests.

1.1. OBJECTIVES OF THIS DOCUMENT

This *HowTo* provides a short guide to the use of the Classad language. It summarises the main goals this language has been designed to meet and describes the rules governing it.

1.2. APPLICATION AREA

Users of the DataGrid WMS software can refer to this document for learning hot to build jobs descriptions for submitting their applications.

1.3. APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Applicable documents

[A1] Matchmaking: Distributed Resource Management for High Throughput Computing Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing,
July 28-31, 1998, Chicago, IL.

[A2] Matchmaking Frameworks for Distributed Resource Management Ph.d Dissertation of Rajesh Raman, October 2000

[A3] JDL Attributes - DataGrid-01-NOT-0101-0_4 – Decemeber 17, 2001, Rome
(http://www.infn.it/workload-grid/docs/DataGrid-01-NOT-0101-0_4.pdf)

Reference documents

[R1] WP1 - WMS Software Administrator and User Guide – DataGrid-01-TEN-0118-0_0 –
December 17, 2001, Rome
(http://www.infn.it/workload-grid/docs/DataGrid-01-TEN-0101-03-Document.pdf)

1.4. DOCUMENT EVOLUTION PROCEDURE

The content of this document will be subjected to modification according to the following events:

- Comments received from WP1 and/or other DataGrid Project members,
- Changes/evolutions/additions to the Job Description Language.

1.5. TERMINOLOGY

Definitions

Condor	Condor is a High Throughput Computing (HTC) environment that can manage very large collections of distributively owned workstations

Glossary

CE	Computing Element
classad	Classified advertisement
GIS	Grid Information Service (aka MDS)
JDL	Job Description Language
JSS	Job Submission Service
LRMS	Local Resource Management System
MDS	Metacomputing Directory System (aka MDS)
PM	Project Month
RB	Resource Broker
SE	Storage Element
TBC	To Be Confirmed
TBD	To Be Defined
UI	User Interface
WMS	Workload Management System
WP	Work Package

2. EXECUTIVE SUMMARY

This document comprises the following main sections:

Section 3: Classified Advertisement Language

Provides a detailed description of properties and rules governing the ClassAd language.

Section 4: Describing Entities

Describes how entities (i.e. jobs and resources) can be described using the ClassAd language features.

Section 5: Annexes

Describes in detail the set of JDL attributes that are meaningful and are used for describing jobs together with their requirements within the DataGrid project.

3. CLASSIFIED ADVERTISEMENT LANGUAGE

3.1. OVERVIEW

The Job Description Language (JDL) adopted within the DataGrid WMS is the Classified Advertisement language defined by the Condor Project (see at the URL http://www.cs.wisc.edu/condor/classad/) for describing jobs, workstations, and other resources.

The classad language is a simple expression-based language that enables the specification of many interesting and useful resource and customer policies facilitating the operation of identification and ranking of compatible matches between resources and customers. It has as its major goal to allow the easy matching between resources and requests, to correctly have jobs executed on the Grid.

The Classad language is therefore the language "spoken" by the WMS components that are directly involved in the job submission process i.e. UI, RB and JSS. It is based on simple descriptional statements like, for instance,

```
UserContact = "Mario.Rossi@esa.int";
RetryCount = 5;
```

therefore in the general format

```
attribute = value;
```

where values can be of different types : numeric, strings, booleans, timestamps etc.

Some of these attributes are used to describe the technical characteristics of the job to be submitted to pass information to the RB, e.g. the Executable and the standard input StdInput attributes:

```
Executable = "sim.exe";
StdInput = "dataset.in"
```

while some other attributes are used to specify requirements for a computing element which is supposed to be found by the RB and to be the executor of the given Job, e.g. the Requirements and Rank attributes, specifying a given set or constraints and preferences on the executor node to be found. The requirements statements syntax looks as follows

```
Requirements = other.OpSys == "RH 6.2" && other.Arch == "INTEL";
```

We notice here that we have introduced the prefix "`other.`" before the attribute name, that specifies that the given statement (`OpSys = "RH 6.2"`, for instance) refers to the possible candidate machine. When not specified the prefix assumes the default value "`self.`", indicating that the statement refers to the job characteristics description.

Therefore the main goals the ClassAd language has been designed to meet can be summarised by the following points:

- **Symmetric:** a key requirement of the advertisement language is to be symmetric with respect to both providers and requesters. The implication of this requirement is that the advertisement language must be powerful and flexible enough to subsume the functionality of traditional resource description and resource selection languages commonly found in conventional resource management systems and also provide the dual properties of customer description and customer selection. This means that both customer and resources (i.e. jobs and computing elements) can be described through classads that can contain constraints respectively on resources and customers.
- **Semi-structured:** the proscription of centralised control (and hence centralised schema management) has naturally suggested the use of a semi-structured model as the basis of the description language. Semi-structured data models (such as XML) are finding widespread acceptance due to their flexibility in managing heterogeneous and distributed information.
- **Declarative:** the advertisement language is required to be declarative rather than procedural. By this it is meant that advertisements should describe notions of compatibility qualitatively rather than specifying a procedure for determining compatibility.
- **Simple:** it is extremely important for an advertising language to be simple both syntactically and semantically. A complex specification language is less amenable to efficient and correct implementation. Complex languages also compound the process of specifying and understanding policies, making both manual and automatic policy management difficult.
- **Portable:** the language must be amenable to efficient implementation on various hardware and software platforms. Thus, it is not reasonable to introduce language features that require specific features of the host architecture that may not be widespread.

As already mentioned the central construct of the language is the *classad*, which is a record-like structure composed of a finite number of distinctly named expressions, as illustrated in Figure 1. Each named expression is called an *attribute*. Classads are used as attribute lists by entities to describe their characteristics, constraints and preferences. Since whole expressions (and not just scalar values) are bound to attribute names, classads can naturally accommodate the predicate-like constraints used by principals to define their policy requirements. Similarly, preferences are specified as expressions that are evaluated to numeric values denoting the “goodness" of candidate matches.

```
[
Executable = "WP1testF";
StdOutput = "sim.out";
StdError = "sim.err";
InputSandbox = {"/home/datamat/sim.exe", "/home/datamat/DATA/*"};
OutputSandbox = {"sim.err","sim.err","testD.out"};
Rank = other.TotalCPUs * other.AverageSI00;
Requirements = other.LRMSType == "PBS" \
&& (other.OpSys == "Linux RH 6.1" || other.OpSys == "Linux RH 6.2") && \
self.Rank > 10 && other.FreeCPUs > 1;
RetryCount = 2;
Arguments = "file1";
```

```
InputData = "LF:test10099-1001";
ReplicaCatalog = "ldap://sunlab2g.cnaf.infn.it:2010/rc=WP2 INFN Test Replica
Catalog,dc=sunlab2g, dc=cnaf, dc=infn, dc=it";
DataAccessProtocol = "gridftp";
OutputSE = "grid001.cnaf.infn.it";
]
```

Figure 1: A classad describing a submitted job

The classad language differentiates between *expressions* and *values*: expressions are evaluable language constructs obtained by parsing valid expression syntax, whereas values are the results of evaluating expressions. The classad language employs *dynamic typing*, so only values (and not expressions) have types. The language has a rich set of types and values which includes many traditional values (numeric, string, boolean), non-traditional values (timestamps, time intervals) and some esoteric values, such as **undefined** and **error**. **Undefined** is generated when an attribute reference cannot be resolved, and **error** is generated when there are type errors. In a sense, all classad operators are *total functions*, since they have a defined semantics for every possible operand value, facilitating robust evaluation semantics in the uncertain semi-structured environment.

Classads may be nested to yield a hierarchical name-space, in which case *lexical scoping* is used to resolve attribute references. An attribute reference made from either customer or resource classad of the form "`other`.*attribute-name*" refers to an attribute named *attribute-name* of the other advertisement. In addition, every classad has a built-in attribute `self` which evaluates to the innermost classad containing the reference, so the reference "`self`.*attribute-name*" refers to an attribute of the same classad containing the reference. If neither self nor other is mentioned explicitly, the evaluation mechanism assumes the `self` prefix. For example, in the Requirements of the job-ad in Figure 1, the sub-expression `other.FreeCPUs > 1` expresses the requirement that the target machine has more than one free CPU for running the job. On the other hand the expression `self.Rank > 10` could also have been written `Rank > 10`.

A reference to a non-existent attribute evaluates to the constant **undefined**. Most operators are "strict" with respect to this value, meaning with this that if either operand is **undefined**, the result is **undefined**. In particular, comparison operators are strict, so that

`other.MinPhysicalMemory > 32,`

`other.MinPhysicalMemory == 32,`

`other.MinPhysicalMemory != 32,`

and

`!(other.MinPhysicalMemory == 32)`

all evaluate to **undefined** if the target classad (i.e. the classad describing the resource, whose attributes are referred with the "`other.`" Prefix) has no `MinPhysicalMemory` attribute. The Boolean operators **||** and **&&** are non-strict on both arguments, so that

`other.MaxRunningJobs >= 10 || other.MaxTotalJobs >= 100`

evaluates to **true** whenever either of the attributes `MaxRunningJobs` or `MaxTotalJobs` exists and satisfies the indicated bound. There are also non-strict operators **is** and **isnt**, which always return boolean results (not **undefined**), allowing explicit comparisons to the constant **undefined** as in:

`other.MinPhysicalMemory is undefined || other.MinPhysicalMemory < 32`

3.2. TYPES AND VALUES

We can view types as a partitioning of the universe of values in the language, where every partition is non-empty. To aid in the unambiguous definition of language semantics, we define fixed internal implementation representations for certain values (such as numbers), while leaving representations of other values unspecified. Values in the classad language can be grouped in two main categories, *literals* and *aggregates* and may be one of the following types:

Literals

- **Undefined:** the undefined type has exactly one value: the **undefined** value. As its name suggests, the **undefined** value represents incomplete or unknown evaluation results due to absence of information. The adoption of a semi-structured data model *requires* the inclusion of an **undefined** (or similar) value for robust evaluation semantics.
- **Error:** the error type has exactly one value: the **error** value. Similar to the **undefined** value, the **error** value plays an important part in securing robust evaluation semantics in semi-structured environments. While the **undefined** value represents missing information, the **error** value represents incorrect or incompatible information, and is usually generated when operators are supplied with values that are outside the domains of their operands. For example, the quotient of a number and a string is **error**.
- **Boolean:** there are exactly two distinct boolean values: **false** and **true**. Unlike their C and C++ counterparts, boolean values are not considered numeric values, and therefore cannot be directly used in numeric expressions.
- **String:** string values are finite sequences of non-zero 8-bit ASCII characters (e.g., "foo", "bar", etc.). There is no *a priori* limit of the length of string values.
- **Integer:** integer values are signed 32-bit two's complement numbers (e.g., 314, -17, 0, etc.). May be expressed in hex or octal (e.g., 0xff, 0777, etc.)
- **Real:** real values are IEEE-754 double precision numbers (e.g., 3.14159, 2.781, etc.).
- **Absolute Time:** absolute time values are non-negative discrete integral values recording the number of seconds elapsed between the UNIX epoch (i.e. 1 January 1970) and the timestamp represented by the value (e.g. “Thu Dec 20 18:21:07 2001 (CDT) -06:00”). Absolute time values must be able to represent the largest integer value as a valid timestamp.
- **Relative Time:** relative time values are discrete integral values that represent time intervals in seconds (e.g. “18:21:32”, “3d19:49:15”). Relative time values may be negative or zero. The cardinality of the relative time value set must be at least as large as the set of integer values.

Aggregates

Classad: classad values are sets of *identifier, expression* equalities separated by semicolons and enclosed between square brackets, where each identifier is distinct (ignoring case) from the others, e.g.

```
[ OpSys = "Solaris8" ]
[ FreeCPUs  = 4; MaxRunningJobs = 100 ]
```

Identifiers are strings of alphanumeric characters and underscores, which begin with non-numeric characters. Classad values additionally indicate (directly or indirectly) the presence of a *parent classad* (or parent scope), which is the closest enclosing classad. If a classad is not lexically nested, it is called a *toplevel* (or root) classad, and its corresponding value does not have a parent scope component.

- **List:** list values are finite sequences of expressions.

3.3. EXPRESSIONS AND EVALUATION SEMANTICS

The majority of the classad language is straightforward and familiar, with some modest extensions. Most of the subtlety of the classad language lies in the treatment of attribute references, which operate in a lexical scoping formalism, but may also explicitly traverse the hierarchical classad namespace during an evaluation to access an attribute. All expression evaluations occur in the context of a given classad, which may be nested arbitrarily deep inside other classads. However, for any given expression evaluation, there is a single unique outermost classad that is not nested. We designate this classad the *root* (or *toplevel*) classad.

3.3.1. ClassAd Expressions

A classad is constructed with the classad construction operator [], and it is sequence of zero or more pairs (*name,expression*) separated by semi-colons as shown in the syntax schema below:

[$name_0$ = $expr_0$; $name_1$ = $expr_1$; . . . . ; $name_n$ = $expr_n$]

Each $name_i$ is a unique identifier and each $expr_i$ is an expression. A classad expression evaluates to a classad value. Every classad value has three implicit attributes: `self`, `parent` and `root`. These attributes are reserved in the concrete syntax and therefore may not be used as any of the $name_i$.

Each classad defines a scope from which attributes may be looked up. Classads may be arbitrarily nested

(e.g. `[ foo=10; bar=[adr=20; adl=30] ]`).

3.3.2. List Expressions

A list is constructed with the list construction operator {} and it is a sequence of zero or more expressions separated by commas as illustrated below:

{$expr_0$, $expr_1$, ... , $expr_n$ }

A list expression evaluates to a list value, which can be later used as an array through use of the subscript operator, moreover they can be arbitrarily nested (e.g., `{10,[foo={10,3,5}],{17,[bar=3]}}`).

3.3.3. Literals

Literals are atomic expressions that directly evaluate to scalar values (i.e., non-classad and non-list values). In this sense, literals directly represent the values that they evaluate to. Examples of literal expressions for values of the various types are provided below. With the exception of string literals, all literals are case insensitive.

- **Undefined**: undefined
- **Error**: error
- **Boolean**: false, true
- **String**: "foo", "bar\n\t" (C-style escapes are supported.)
- **Integer**: 10, 0xff (Hex), 0600 (Octal)
- **Real**: 3.141, 6.023e23, 2K (i.e., 2048.0). The suffixes B, K, M, G and T representing scale factors of 2^0, 2^{10}, 2^{20}, 2^{30} and 2^{40} are all supported.
- **Absolute Time**: 'Thu Aug 17 18:21:07 2000 (CDT) -06:00'
- **Relative Time**: '18:21:32', '3d19:49:15'

3.3.4. Operations

Operations are expressions that combine other expressions by means of unary, binary and ternary operators. The operators are essentially those of the C language, with certain operators excluded (e.g., pointer and dereference operators) and others added (e.g., non-strict comparison). Thus, a rich set of arithmetic, logic, bitwise and comparison operators are defined. The set of supported operators and their relative precedences are summarized in Figure 3. In the following specification of operator semantics, it is to be assumed that unless otherwise specified, operators are strict with respect to the **undefined** and **error** values in all places, with **error** taking precedence over **undefined**.

Operator class	Operators	Associativity
Primary	[]	Left to right
Unary	- + ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< <= > >=	Left to right
Equality	== != is isnt	Left to right
Bitwise AND	&	Left to right

Operator class	Operators	Associativity
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left

Table 1 Classad language operators in decreasing order of precedence

Additionally, since most operators are meaningfully defined only over certain values, we define operations to evaluate to **error** when values outside the domain of an operator are supplied as operands. In other words, unless otherwise specified, the following implicit rules must be applied (in order) to all following specifications:

- **Strictness Rule:** if any operand to an operator is **undefined** (**error**), the resulting value of the operation is also **undefined** (**error**). If both **undefined** and **error** are simultaneously supplied to an operator, the result is **error**.
- **Domain Rule:** if the operands to the operator are outside the operator's domain, the resulting value of the operation is **error**.

We now informally describe the behaviours of operators in the classad language.

3.3.4.1. Arithmetic Operators

All arithmetic operators are binary, and follow both Strictness and Domain Rules. The domain for arithmetic operators is numeric values, i.e., the integer and real values. With the inclusion of the following rules, arithmetic in the classad language occurs in "the natural way".

1. If the divisor is zero in the case of the division (/) and remainder (%) operators, the evaluation result is **error**.
2. If one operand is integer and the other is real, the integer operand is promoted to a real, and the evaluation proceeds as a computation of real numbers. Unless the expression violates any of the previous rules, the type of the evaluation result is real.

3.3.4.2. Comparison Operators

All comparison operators are binary and, with the exception of the `is` and `isnt` operators, follow both Strictness and Domain Rules. The following rules define the behavior of strict comparison.

1. Only values of the same type may be compared. The only exception to this rule is that integers and reals may be compared: the integer is promoted to a real, and comparison proceeds as with real values.
2. Only scalar values may be compared. Comparison of aggregate values (i.e., classads and lists) results in **error**.

3. (Boolean specialization) The **false** value is defined to be less than the **true** value.
4. (String specialization) All string comparisons are case insensitive, so "FOO", "fOO" and "fOo" are all equivalent. Strings are ordered lexicographically, ignoring case.
5. (Absolute time specialization) An absolute time value is defined to be less than another if the timestamp it represents temporally precedes the timestamp represented by the other comparand.
6. (Relative time specialization) Shorter intervals are less than longer intervals.

The non-strict comparison operators `is` and `isnt` implement the "is identical to" and "is not identical to" predicates, and can therefore be used to test if given values are **undefined** or **error**. By definition, these operators follow neither Strictness nor Domain Rules, these operators *always* evaluate to **true** or **false** (e.g. `undefined is 10` evaluates to `false`, while `error is error` evaluates to `true`).

The following rules, when applied in order, summarize the behavior of the `is` operator (the `isnt` operator is simply the boolean negation of the `is` operator):

1. If the types of the two comparands differ, the result of the comparison is **false**.
2. If the type of one comparand is **undefined** (**error**), the result of the operation is true if the other comparand is also **undefined** (**error**), and false otherwise.
3. Comparison of aggregate values is not allowed, so the result of the `is` operator is **false** if either operand is an aggregate value.
4. Comparison of string values is case sensitive. This behavior is different than that of the strict comparison operators.
5. Otherwise, the `is` operator behaves exactly like the equals comparison operator (==).

3.3.4.3. Bitwise Operators

The bitwise operators follow both Strictness and Domain rules, and are applicable only to integer values. The operators behave identically to their counterparts in the Java programming language.

3.3.4.4. Logic Operators

The logic operators OR (||) and AND (&&) are non-strict operators, and therefore do not follow the implicit Strictness Rule. Instead the operators follow the truth tables supplied below (**Table 2**), in which T, F, U and E stand for **true**, **false**, **undefined** and **error** respectively. If any operand does not evaluate to a boolean, undefined or error value, the result of the operation is error.

AND	F	T	U	E
F	F	F	F	E
T	F	T	U	E
U	F	U	U	E

OR	F	T	U	E
F	F	T	U	E
T	T	T	T	E
U	U	T	U	E

NOT	
F	T
T	F
U	U

E	E	E	E	E

E	E	E	E	E

E	E

Table 2 Logic operators truth tables

3.3.4.5. Miscellaneous Operators

3.3.4.5.1. The Subscript Operator

The subscript operator is a binary operator that follows both Strictness and Domain Rules. It requires one list type operand (i.e., an array), and one integer type operand (i.e., an index). If the supplied index is not a non-negative integer less than the length of the array, the operation evaluates to **undefined**. Otherwise, the result of the operator is the value of the index'th expression in the array (with zero based indexing like in C/C++; e.g. `{10, 17*2, 30}[1]` **=>** `34`).

3.3.4.5.2. The Conditional Operator

The conditional operator is the only ternary operator in the classad language. It follows the Strictness and Domain rules only with respect to its first operand (the condition), which is required to be boolean. The result of the evaluation is the value of the second operand (the true consequent) if the condition evaluates to **true**, and the value of the third operand (the false consequent) if the condition evaluates to **false**.

The conditional operator is not strict for the two consequents (e.g. `true?10:undefined` evaluates to `10`, and `false?error:"foo"` evaluates to `"foo"`).

3.3.5. Attribute References

Attribute references in the classad language are similar to both variable references in programming languages like C and C++, and filenames in the UNIX filesystem. In the following description of the three variants of attribute reference expressions, *attr* denotes a case-insensitive identifier and *expr* denotes an arbitrary expression:

attr

This attribute reference variant has two possible behaviors. If *attr* is one of the following special built-ins, the reference evaluates to certain predefined values.

1. The *self* attribute reference evaluates to the classad that serves as the current scope of evaluation.
2. The *root* attribute reference evaluates to the classad that serves as the root of the evaluation.
3. The *parent* (or *super*) attribute reference evaluates to the classad that is the lexical parent of the current evaluation scope. If the current evaluation scope is the root scope, the *parent* attribute reference evaluates to **undefined**.

If the reference is not one of the above three special built-ins, the reference evaluates to the value of the expression bound to the attribute named *attr* in the closest enclosing scope. (The obtained expression must be evaluated in the same scope that it was found.) If no such attribute is found, the reference evaluates to the **undefined** value. Some examples are reported below (evaluated expressions are the ones in bold):

Top-level ClassAd	Value
`[a=1;b= **a** ]`	1
`[a=2;b=[c=1;d= **a** ]]`	2
`[a=2;b=[c=1;d= **a+f** ];e=[f=10]]`	undefined
`[a=3;b=[c=1;d=[e=5;f= **a+c+e** ]]]`	9
`[a=3;b=[a=2;c=1;d=[e=5;f= **a+c+e** ]]]`	8

.attr

This attribute reference variant evaluates to the value of the expression bound to the name *attr* in the root scope, when evaluated in the *root* scope. If the *root* scope does not contain an attribute named *attr*, the value of the reference is **undefined**. Some examples are reported below (evaluated expressions are the ones in bold):

Top-level ClassAd	Value
`[a=2;b=[a=1;d= **.a** ]]`	2
`[a=3;b=[a=1;d=[a=5;f= **a+.a** ]]]`	8
`[a=2;b=[c=1;d= **.c** ]]`	undefined

expr.attr

This variant first evaluates the expression *expr*, which must evaluate to a classad. (If this expression evaluates to **undefined**, the value of the entire reference is **undefined**. Otherwise, if the value is not a classad, the value of the reference is **error**.) The value of the reference is the value of the expression bound to the attribute named *attr* in the closest enclosing scope beginning with the classad scope identified by *expr*. As with previous variants the identified expression must be evaluated in the scope it was obtained from, and if no such expression exists, the value of the reference is **undefined**. Some examples are reported below (evaluated expressions are the ones in bold):

Top-level ClassAd	**Value**
`[a=1;b= [c=5].c ]`	5
`[a=1;b= [c=5].a ]`	1
`[a=1;b=[a=2;c=[b=.a];d= .b.c.a ]`	2
`[a=1;b=[a=2;c=[b=.a];d= .b.c.b ]`	1
`[a=1;b=[a=2;c=[b=a];d= .b.c.b ]`	2
`[a=1;b=[a=2;c=[b=.a];d= .a.b.a.b ]`	error
`[a=1;b=[c=2];d=[super=.b]].d.c`	2
`[a=1;b=[a=7;c= super.a ]]`	1

In the next Figure 2 is reported another example of attribute references that comprises most of the cases we have dealt with:

```
[
 adl=[
      other = .adr.self;
      self  =[ Owner = "Ms. Foo";
               Arch = "INTEL";
               MemorySize = 32M;
               Requirements = other.Owner != "foo"
              ];
      ];
 adr=[
      other = .adl.self;
      self  = [ Owner = "Mr. Bar";
                MemorySize = 16M;
                Requirements = (other.Arch=="INTEL") &&
                  (other.MemorySize > self.MemorySize)
              ]
      ]
 ]
```

Figure 2 Attribute References – 1

Finally, consider the following classad

```
[
   a = 17;
   b = "foo";
   c = { "x", "y", 3*a };
   d = [
          a = 23;
          b = 15;
          c = []
          d = .a;
       ]
   e = '00:15:00';
]
```

Figure 3 Attribute References – 2

Hereafter are reported some expressions and their resulting values when evaluated in the context of the classad in Figure 3.

Expression	Result
`a`	`17`
`x`	`undefined`
`x+10`	`undefined`
`x || true`	`true`
`d.a`	`23`
`d.b`	`15`
`d.self`	`[ a = 23 ; b = 15 ; c = [] ; d = .a ]`
`d.c`	`[]`
`d.c.parent`	`[ a = 23 ; b = 15 ; c = [] ; d = .a ]`
`d.b.a`	`error`
`d.parent.c`	`{ "x", "y", 3*a }`
`d.parent.c[2]`	`51` *(17*3)*
`d.d`	`17`
`e*4`	`'01:00:00'`

Table 3 Expression evaluation

3.3.6. Circular Expression Evaluation

It is trivially possible for expressions in the classad language to refer to each other in a manner that would lead to an infinite loop during expression evaluation. For example, in the classad `[a=b; b=a]`, it is not possible to determine the value of either attribute. The classad language defines that circular expression evaluation result in the **undefined** value. Two examples are reported below:

- Circularities in expression evaluation: `[b = a; a = b].a` evaluates to `undefined`
- Circularities in scoping: `[a=[super=.b]; b=[super=.a]].x` evaluates to `undefined`

3.3.7. Function Calls

The classad language provides a number of built-in utility functions to perform tasks such as string pattern matching, obtaining the current time of day, converting values from type to another and testing value types.

User-defined functions may not be defined. The syntax of a function call is "*name*(arg_0 ; arg_1 ; . . . . ; arg_n)"; for example in the context of the classad of Figure 3, we have `strcat(b, "bar", a)` that evaluates to `"foobar17"`.

As with operators, most functions are strict with respect to **undefined** and **error** on all arguments. However, some functions are non-strict, and these exceptions are noted. The name of the function is not case sensitive. A comprehensive list of functions and their behaviors is provided hereafter:

Type predicates (Non-Strict)

- IsUndefined(V) True iff V is the **undefined** value.
- IsError(V) True iff V is the **error** value.
- IsString(V) True iff V is a string value.
- IsList(V) True iff V is a list value.
- IsClassad(V) True iff V is a classad value.
- IsBoolean(V) True iff V is a boolean value.
- IsAbsTime(V) True iff V is an absolute time value.
- IsRelTime(V) True iff V is a relative time value.

List Membership

- Member(V,L) True iff scalar value V is a member of the list L.
- IsMember(V,L) Like Member, but uses `is` for comparison instead of ==. Not strict on first argument.

Time Queries

- CurrentTime() Get current time (absolute time)
- TimeZoneOffset() Get time zone offset as a relative time
- DayTime() Get current time as relative time since midnight.

Time Construction

- MakeDate(M,D,Y) Create an absolute time value of midnight for the given day. M can be either numeric or string (e.g., "jan").
- MakeAbsTime(N) Convert numeric value N into an absolute time (number of seconds past UNIX epoch).
- MakeRelTime(N) Convert numeric value N into a relative time (number of seconds in

interval).

Absolute Time Component Extraction

- GetYear(A) Get integer year (A=absolute time)
- GetMonth(A) 0 = *jan; . . . .;* 11 = *dec*
- GetDayOfYear(A) 0 . . . .365 (for leap year)
- GetDayOfMonth(A) 1 31
- GetDayOfWeek(A) 0 6
- GetHours(A) 0 23
- GetMinutes(A) 0 59
- GetSeconds(A) 0 . . . .61 (for leap seconds)

Relative Time Component Extraction

- GetDays(R) Get days component in the interval (R= relative time)
- GetHours(R) 0 23
- GetMinutes(R) 0 59
- GetSeconds(R) 0 59

Time Conversion

- InDays(T) Convert time value into number of days
- InHours(T) Convert time value into number of hours
- InMinutes(T) Convert time value into number of minutes
- InSeconds(T) Convert time value into number of seconds

String Functions

- StrCat(V1, . . . , Vn) Concatenates string representations of values V1 through Vn
- ToUpper(S) Upcases string S
- ToLower(S) Downcases string S
- SubStr(S,offset [,len]) Returns substring of S. Negative offsets and lengths count from the end of the string.
- RegExp(P,S) Checks if S matches pattern P (both args must be strings).

Type Conversion Functions

- Int(V) Converts V to an integer. Time values are converted to number of seconds, strings are parsed, bools are mapped to 0 or 1. Other values result in **error**
- Real(V) Similar to Int(V), but to a real value.
- String(V) Converts V to its string representation

- Bool(V) Converts V to a boolean value. Empty strings, and zero values converted to **false**; non-empty strings and non-zero values converted to **true**.
- AbsTime(V) Converts V to an absolute time. Numeric values treated as seconds past UNIX epoch, strings parsed as necessary.
- RelTime(V) Converts V to an relative time. Numeric values treated as number of seconds, strings parsed as necessary.

Mathematical Functions

- Floor(N) Floor of numeric value N
- Ceil(N) Ceiling of numeric value N
- Round(N) Rounded value of numeric value N

4. DESCRIBING ENTITIES

We will provide in this section some examples of job and computational resources descriptions made through the presented classad language. Our goal is not only to show the way an entity can publish its detailed characteristics but also to demonstrate the flexibility of the mechanism in expressing fairly sophisticated policies.

As already mentioned in the previous sections, the classad language is extensible and semi-structured, hence each job/resource owner/administrator can freely include in its advertisements all new attributes that are necessary or relevant for its branch-specific description. Anyway, since advertisements are made to be used in a matchmaking process by the resource management system, all the entity description shall conform to a set of conventions (a *protocol*), which binds meanings to certain attributes that will be used for special purposes. For example, in our framework we will define that in any classad shall contain the attributes named `Requirements` and `Rank` that will be respectively treated as the constraints and preferences expressed by the advertising entity. Note that in such a context it is interest of the involved parties to provide the better-detailed description as possible in order to obtain the best match (see Annex 5 at the end of this document for a preliminary list of common attributes that can be used to build entities descriptions for Datagrid purposes).

4.1. CE ACCESS CONTROL

Figure 4 shows a classad that describes a CE and demonstrates the way to express access control on a resource by means of the language features. The `Requirements` attribute indicates that the CE only accepts to run applications from authorised members i.e. members whose certificate subject is listed in the grid-mapfile of the CE.. The `Rank` expression states that jobs with a required lower number of retrials on failure have higher priority than other jobs.

```
[
      CEId = "lxde01.pd.infn.it:2119/jobmanager-lsf-grid01";
      GlobusResourceContactString = "lx01.pd.infn.it:2119/jobmanager-lsf";
      GRAMVersion = "1.71";
      Architecture = "INTEL";
      OpSys = "RH 6.1";
      MinPhysicalMemory = 256;
      MinLocalDiskSpace = 100;
      TotalCPUs = 4;
      FreeCPUs = 1;
      TotalJobs = 15;
      RunningJobs = 4;
      IdleJobs = 11;
      MaxTotalJobs = 1000;
      MaxRunningJobs = 1000;
      WorstTraversalTime = 2502;
      EstimatedTraversalTime = 131;
      AverageSI00 = 23;
```

```
    MinSI00 = 13;
    MaxSI00 = 30;
    AuthorizedUser = {"/C=IT/O=INFN/L=Padova/CN=Mario
                      Rossi/Email=mario.rossi@pd.infn.it",
                      "/C=IT/O=INFN/L=Milano/CN=Ugo Bianchi
                      /Email=Ugo.Bianchi@mi.infn.it",
                      "/O=Grid/O=UKHEP/OU=hep.ph.ac.uk/CN=Tom Scott"};
    RunTimeEnvironment = {"CMS3.2"," EO4.2"};
    AFSAvailabe = True;
    OutboundIP = True;
    InboundIP = False;
    QueueName = "grid01";
    LRMSType = "LSF";
    LRMSVersion = "4.0";
    Rank = 10 - other.RetryCount;
    Requirements = Member(AuthorizedUser, other.CertificateSubject);
]
```

Figure 4 CE Access Control

It is important to remark that the construction of CEs descriptions in the ClassAd language is a task performed automatically by the Resource Broker during the matchmaking process and it is hence never done by the user. This example of resource classad has been reported here only to show the symmetry property of the ClassAd language.

4.2. RESOURCE CONSTRAINTS

Figure 5 describes a job that has the policy of running only on INTEL machines with sufficient memory space (in Mbytes), running the LINUX or the Solaris operating system on which outbound connectivity is allowed (e.g. the job can "initiate" a data transfer, sending and/or receiving data to/from a remote Internet node). In addition, the `Rank` expression in the job classad expresses a preference for running on CEs having the greater number of free CPUs.

```
[
   CertificateSubject = "/O=Grid/O=UKHEP/OU=hep.ph.ac.uk/CN=Tom Scott";
   Executable = "WP1testF";
   Arguments = "datafile1.in 5.56 1024";
   StdInput = "sim.dat" ;
   StdOutput = "sim.out" ;
   StdError = "sim.err" ;
   InputSandbox = {"/home/fpacini/DATA/datafile1.in" ,
                   "/home/fpacini/DATA/sim.dat",
                   "/home/fpacini/exe/WP1testF",
                   "/home/fpacini/DATA/file2"};
   OutputSandbox = {"sim.err","sim.out"};
   InputData = {"LF:test10096-0009" , "LF:test100960010",
               "PF:testbed002.cern.ch/home/flavia/ffiles/test10096-
               0011"};
   ReplicaCatalog = "ldap://sunlab2g.cnaf.infn.it:2010/rc=WP2 INFN Test
                    Replica Catalog,dc=sunlab2g, dc=cnaf, dc=infn,
                    dc=ita" ;
   DataAccessProtocol = "gridftp";
   OutputSE = "lx11.hep.ph.ic.ac.uk";
   RetryCount = 6;
   Rank = other.FreeCPUs;
   Requirements = other.Architecture == "INTEL" && (other.OpSys == "RH
                  6.2" ||  other.OpSys == "Solaris 2.6") &&
                  other.MinPhysicalMemory >= 200  && other.OutboundIP ==
                  TRUE;
]
```

Figure 5 Resource Constraints 1

We now present another example of specification of constraint and preferences on resources. In the example illustrated in Figure 6, the customer requires a CE being an INTEL machine running the LINUX RH 6.1 operating system on which the is installed the EO4.2 run-time environment. Moreover the local resource management system is required to be PBS. The `Rank` expression in the job classad expresses a preference for running on CEs having a greater number of allowed maximum running jobs and AFS installed.

```
[
      Executable = "/opt/edg/WP1testC";
      StdInput = "sim.dat" ;
      StdOutput = "sim.out" ;
      StdError = "sim.err" ;
      InputSandbox = {"/home/fpacini/DATA/file1",
                      "/home/fpacini/DATA/sim.dat",
                      "/home/fpacini/DATA/file2"};
      OutputSandbox = {"sim.err","sim.out","datafile1.out"};
      InputData = {"PF:testbed001.cern.ch/home/ffiles/test10096-0009",
                  "PF:testbed002.cern.ch/home/ffiles/test10096-0011"};
      DataAccessProtocol = "file";
      RetryCount = 3;
      Rank = other.MaxRunningJobs + (other.AFSAvailabe == True ? 10 : 5);
      Requirements = other.Architecture == "INTEL" && other.OpSys == "RH
                      6.1 && Member(other.RunTimeEnvironment , "EO4.2")  &&
                      other.LRMSType == "PBS";
]
```

Figure 6: Resource Constraints 2

5. ANNEXES

5.1. JDL ATTRIBUTES

The JDL is a fully extensible language (i.e. it does not rely on a fixed schema), hence the user is allowed to use whatever attribute for the description of a job without incurring in errors. Anyway only a certain set of attributes (that we will refer to as “supported” attributes) can be taken into account by the WMS components for scheduling a submitted job. Indeed in order to be actually used for selecting a resource, an attribute used in a job class-ad needs to have a correlation with some characteristic of the resources that are published in the GIS (aka MDS).
The “supported” attributes, their meaning and the way to use them to describe a job are dealt in detail in document [A3] also available at the following URL:
http://www.infn.it/workload-grid/docs/DataGrid-01-NOT-0101-0_4.pdf.