

DataGrid

JOB DESCRIPTION LANGUAGE HowTo



Document identifier: **DataGrid-01-TEN-0102-0_0**

Date: **28/05/2001**

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	28/05/2001	
Verified by	Stefano Beco	Datamat SpA	28/05/2001	
Approved by				

Document Log

Issue	Date	Comment	Author
0_0	28/05/2001	First draft	Fabrizio Pacini

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files
Word 97	DataGrid-01-TEN-0102-0_0-Document.doc
Acrobat Exchange 4.0	DataGrid-01-TEN-0102-0_0-Document.pdf



CONTENT

1. INTRODUCTION.....	4
1.1. OBJECTIVES OF THIS DOCUMENT.....	5
1.2. APPLICATION AREA	5
1.3. APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS.....	5
1.4. DOCUMENT EVOLUTION PROCEDURE	6
1.5. TERMINOLOGY	6
2. EXECUTIVE SUMMARY.....	7
3. CLASSIFIED ADVERTISEMENT LANGUAGE.....	8
3.1. OVERVIEW	8
3.2. TYPES AND VALUES	10
3.3. EXPRESSIONS AND EVALUATION SEMANTICS	11
3.3.1. <i>ClassAd Expressions</i>	11
3.3.2. <i>List Expressions</i>	11
3.3.3. <i>Literals</i>	12
3.3.4. <i>Operations</i>	12
3.3.5. <i>Attribute References</i>	16
3.3.6. <i>Circular Expression Evaluation</i>	19
3.3.7. <i>Function Calls</i>	20
4. DESCRIBING ENTITIES	23
4.1. WORKSTATION ACCESS CONTROL	23
4.2. TIME-DEPENDENT RESOURCE PREFERENCE.....	25
4.3. TIME-DEPENDENT RESOURCE CONSTRAINTS.....	26
5. ANNEXES.....	27
5.1. JDL ATTRIBUTES	27

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

WP1 and WP8-9-10.

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] DATAGRID WP1 Task 1.2 Job Description Language HowTo, March 07, Rome
(<http://www.infn.it/workload-grid/docs/classad-howtopdf>)

Reference documents

- [R1] Job Submission User Interface Man Pages - DataGrid-01-TEN-0101-0_1, May 28, 2001, Rome



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 WPs partners,
- 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

classad	Classified advertisement
GIS	Grid Information Service
JDL	Job Description Language
PM	Project Month
RB	Resource Broker
TBD	To Be Defined
TBC	To Be Confirmed
UI	User Interface
WP	Work Package



2. EXECUTIVE SUMMARY

Goal of this *HowTo* is not only to provide a short guide to the use of the Classad language but also to have a feedback on it from the Datagrid application work-packages communities in order to asses if it can be elected as the Datagrid JDL. At this aim, after having familiarised with the Classad language, the reader should try to describe its branch-specific applications in the most detailed way as possible, emphasising detected weaknesses, missing features and aspects needing tailoring and modifications.

3. CLASSIFIED ADVERTISEMENT LANGUAGE

3.1. OVERVIEW

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. The main goals it 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.
- **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.

```
[  
  Type = "Job";  
  QDate = 'Mon Jan 25 10:00:05 2001 (GMT) -06:00';  
  CompletionDate = undefined;  
  Owner = "fpacini";
```

```
Executable = "sim.out";
WantRemoteSyscalls = true;
WantCheckpoint = true;
Iwd = "/usr/fpacini/test";
Args = "17 3200 10";
ImageSize = 31M;
Rank      = other.KFlops/1E3 + other.PhysicalMemory/32;
Constraint = other.Type == "Machine" && other.Arch=="INTEL" &&
             other.OpSys=="SOLARIS251" &&
             other.VirtualMemory > self.ImageSize
]
```

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. The scoping features of the language in the context of the "match evaluation environment" established by the entity in charge of testing matches, result in the semantics that 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 Constraint of the job ad in Figure 1, the sub-expression `other.VirtualMemory > self.ImageSize` expresses the requirement that the target machine have sufficient virtual memory to accommodate the requirements of the job. The expression could also have been written `other.VirtualMemory > ImageSize`.

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.PhysicalMemory > 32,
other.PhysicalMemory == 32,
other.PhysicalMemory != 32,
and
!(other.PhysicalMemory == 32)
```

all evaluate to **undefined** if the target classad has no `PhysicalMemory` attribute. The Boolean operators `||` and `&&` are non-strict on both arguments, so that

```
other.Mips >= 10 || other.Kflops >= 1000
```

evaluates to **true** whenever either of the attributes `Mips` or `Kflops` 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.PhysicalMemory is undefined || other.PhysicalMemory < 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. 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. 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 finite sets of (*identifier*, *expression*) pairs, where each identifier is distinct (ignoring case). 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 *oplevel* (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 *oplevel*) classad.

3.3.1. ClassAd Expressions

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

```
[name0 = expr0 ; name1 = expr1 ; . . . ; namen = exprn ]
```

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:

```
{expr0 , expr1 , . . . , exprn }
```

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

Operator class	Operators	Associativity
Equality	== != is isnt	Left to right
Bitwise AND	&	Left to right
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	OR	F	T	U	E	NOT	
F	F	F	F	E	F	F	T	U	E	F	T
T	F	T	U	E	T	T	T	T	E	T	F
U	F	U	U	E	U	U	T	U	E	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;
              Constraint = other.Owner != "foo"
            ];
  ];
  adr=[
    other = .adl.self;
    self  = [ Owner = "Mr. Bar";
              MemorySize = 16M;
              Constraint = (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) 0365 (for leap year)
- GetDayOfMonth(A) 131
- GetDayOfWeek(A) 06
- GetHours(A) 023
- GetMinutes(A) 059
- GetSeconds(A) 061 (for leap seconds)

Relative Time Component Extraction

- GetDays(R) Get days component in the interval (R= relative time)
- GetHours(R) 023
- GetMinutes(R) 059
- GetSeconds(R) 059

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 `Constraint` and `Rank` that will be respectively treated as the constraints and preferences expressed by the advertising entity. Moreover we will require the advertising parties to include "contact addresses" (`Address` attribute) with their ads. 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 Appendix 5.1 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. WORKSTATION ACCESS CONTROL

Figure 4 shows a classad that describes a workstation and demonstrates the way to express access control on a resource by means of the language features. The `Constraint` attribute indicates that the workstation is never willing to run applications submitted by users "rival" and "hacker", it is always willing to run the jobs of members of the research group, friends may use the resource only if the workstation is idle (as determined by keyboard activity and load average) and others may only use the workstation at night. The `Rank` expression states that research jobs have higher priority than friend's jobs, which in turn have higher priority than other jobs.

```
[
  Type                = "Machine";
  Address              = "<firefox.esrin.esa.it:2590>";
  Activity             = "Idle";
  KeybrdIdle          = '00:23:12'; // h:m:s
  Disk                = 323.4m; // mbytes
  PhysicalMemory      = 64m; // mbytes
  State               = "Unclaimed";
  LoadAvg             = 0.042969;
  Mips                = 104;
  Arch                = "INTEL";
  OpSys               = "SOLARIS251";
  KFlops              = 21893;
  ResourceName        = "firefox.esrin.esa.it";
```



```
Machine           = "firefox.esrin.esa.it";
IsInstructional   = FALSE;
RebootedDaily    = FALSE;
OffHoursOnly     = FALSE;
CkptServer       = "tempest.esrin.esa.it";
IsDedicated      = FALSE;
IsComputeCluster = FALSE;
VirtualMachineID = 1;
VirtualMemory    = 746304;
ConsoleIdle      = 70530;
NumCpus          = 1;
UidDomain        = "firefox.esrin.esa.it";
FileSystemDomain = "firefox.esrin.esa.it";
Subnet           = "193.204.228";
TotalVirtualMemory = 746304;
TotalDisk        = 384802;
EnteredCurrentState = 972068166;
EnteredCurrentActivity = 972206722;
ResearchGrp      = {"fpacini ", "sbeco", "ugrand"};
Friends          = {"pastolfi", "mfonti"};
Untrusted        = {"rival", "hacker"};
Rank             = Member(other.Owner, ResearchGp) ? 10 :
                  Member(other.Owner, Friends) ? 1 : 0;
Constraint       = !Member(other.Owner, Untrusted) &&
                  Rank>=10 ? true : Rank>0 ? LoadAvg < 0.3 &&
                  KeybrdIdle>'00:15' : DayTime(<'8:00' ||
                  DayTime(>'18:00'
```

]

Figure 4 Workstation Access Control

4.2. TIME-DEPENDENT RESOURCE PREFERENCE

Customers may incorporate environment specific information to improve the quality of service delivered to their applications. For example, many of the workstations that are used for instructional purposes exist in computer laboratories that are locked during the night. Thus, it is beneficial for applications to run on these machines after hours, as they will not be preempted by machine owners during this time. Figure 5 describes a job that has the policy of running only on INTEL machines with sufficient memory and disk space, running the LINUX operating system. In addition, the Rank expression in the job classad expresses a preference for running on instructional machines during the night over running on a machine that has been idle for a long time (and is therefore likely to remain unused), which is in turn preferred over running on any other machine.

```
[
  Type           = "Job";
  Address        = "<multira2.datamat.it:2020>";
  Priority       = 23.7;
  CompletionDate = undefined;
  RemoteSyscalls = true;
  Checkpoint    = true;
  QDate         = 'Mon Jan 11 10:53:31 2001 (CST) -06:00';
  Owner         = "fpacini";
  Executable    = "sum.exe";
  Iwd           = "/users/fpacini/exe";
  Args          = "3.141 6.023e23";
  ImageSize     = 42M;
  Rank          = DayTime(>'20:00' && DayTime<'8:00' &&
    other.IsInstructional ? 10 :
    other.KeybrdIdle>'3:00' ? 5 : 0;
  Constraint    = other.Type=="Machine" && other.Arch=="INTEL" &&
    other.OpSys=="LINUX" && other.Disk >= 10M &&
    other.PhysicalMemory >= self.ImageSize
]
```

Figure 5 Time Dependent Resource Preference

4.3. TIME-DEPENDENT RESOURCE CONSTRAINTS

We now present an example in which a customer varies the resource constraint over time. In the example illustrated in Figure 6, the customer waits for up to two hours for a resource with at least one gigabyte of memory. If a match hasn't been found with two hours, the customer downgrades the resource requirement

to a resource that has at least half a gigabyte of memory. The Rank expression states that machines with larger memories are preferred. In this example, the customer will reject machines with less than one gigabyte of memory for the first two hours, hoping for a better match. This policy is therefore fundamentally different from one that merely prefers machines with larger physical memories.

```
[
    Type           = "Job" ;
    Address        = "<multira2.datamat.it:2020>" ;
    CompletionDate = undefined ;
    RemoteSyscalls = true ;
    Checkpoint     = true ;
    QDate         = 'Mon Jan 11 10:53:31 2001 (CST) -06:00' ;
    Owner         = "fpacini" ;
    Cmd           = "run_rendering" ;
    Iwd          = "/users/fpacini/exe" ;
    StdIn        = request.sceneFile ;
    StdOut       = request.outputFile ;
    NumCpus      = 2 ;
    MemoryReqs   = 24.2M ;
    ImageSize    = 42M ;
    ElapsedTime  = CurrentTime( ) - QDate ;
    Rank         = other.PhysicalMemory ;
    Constraint    = other.Type=="Machine" && other.Arch==" SUN4u" &&
                  other.OpSys=="SOLARIS251" &&
                  other.PhysicalMemory >=
                  (ElapsedTime>'2:00' ? 0.5G : 1.0G)
]
```

Figure 6: Time Dependent Resource Constraints

5. ANNEXES

5.1. JDL ATTRIBUTES

The following Table 4 and Table 5 report the recommended set of classad attributes to be used to describe jobs when submitting a request through the Job Submission UI. Table 4 contains job specific attributes, i.e. the ones to be used to describe job characteristics, whilst Table 5 contains resource specific attributes that can be used to express job constraints and preferences, i.e. to build the *Requirements* (i.e. the attribute called Constraint in the rest of this document) the and *Rank* expressions.

The JDL language is fully extensible, hence it is allowed to use whatever attribute for the description of a job, anyway the user should base the specification of job constraints and preferences only on the attributes advised in Table 4 and Table 5 (at least for PM9) since they are the ones that are related with those published in the GIS and can then be taken into account by the RB for the match making process. Those attributes whose support is not yet assured for PM9 delivery are greyed.

Attribute	Meaning
Executable	Executable/command name. The user can specify an executable that lies already on the remote CE. The absolute path, possibly including environment variables, of this file should be specified. The other possibility is to provide a local executable name, which will be staged on the CE. In this case only the file name has to be specified as executable. The absolute path on the local file system executable should be then listed in the <i>InputSandbox</i> attribute expression.
InputData	(Exact content of this attribute is TBC) <ul style="list-style-type: none"> - a logical collection, a list logical collections and/or - a list of logical files and/or - a list of physical files This attribute refers to data used as input by the job; these data are stored in SEs and published in replica catalogues.
StdInput	Standard input of the job. It can be: <ul style="list-style-type: none"> - just a file name (staging required) - absolute path (available on the CE) The same mechanism as described for the <i>Executable</i> attribute can be applied.



Attribute	Meaning
StdOutput	Standard output of the job. The user has to specify just the file name. To have this file staged back on the submitting machine he/she has to list the file name also in the <i>OutputSandbox</i> attribute expression and use the dg-get-job-output command [R1].
StdError	Standard error of the job. The user has to specify just the file name. To have this file staged back on the submitting machine he/she has to list the file name also in the <i>OutputSandbox</i> attribute expression and use the dg-get-job-output command [R1].
OutputSE	URI of the Storage Element where to store the output data. Once specified, this attribute shall be used to build the job requirements expression in order make the RB choose a resource being "attached" with this SE. (e.g. <i>Requirements = ... && Member(OutputSE, other.StorageElements);</i>)
InputSandbox	List of files on the UI local disk needed by the job for running. The listed files are staged from the UI to the remote CE.
OutputSandbox	List of files generated by the job that have to be retrieved. The listed files are transferred on the UI local file system by mean of the dg-get-job-output command [R1].
RetryCount	Number of job submission retrial made by JSS. Default value is 3.
ReplicaCatalog	Replica Catalogue Identifier, i.e. something in the following format: <protocol>://<full hostname> :<port>/<Replica Catalog DN>. This attribute is mandatory if the <i>InputData</i> attribute has been also specified
Rank	a ClassAd Floating-Point expression that states how to rank queues that have already met the Constraints expression. Essentially, rank expresses preference. A higher numeric value equals better rank. The RB will



Attribute	Meaning
	give the job the queue with the highest rank. Default value for this attribute is: <i>-EstimatedTraversalTime</i>
Requirements	Boolean ClassAd expression which uses C-like operators. It represents job requirements on resources. In order for a job to run on a given queue, this Constraint expression must evaluate to true on the given queue. Default value for this attribute is TRUE.

Table 4 classad job specific attributes



JOB DESCRIPTION LANGUAGE HOWTO

Doc. Identifier:

DataGrid-01-TEN-0102-0_0

Date: 28/05/2001

Attribute	Meaning
ResourceManagementType	Defines the type of resource management system (LSF/Condor/...).
ResourceManagementVersion	The version of the local resource management system.
GRAMVersion	the GRAM version.
Architecture	the architecture of the machine or of the machines associated to the queue (we assume that all the machines "belonging" to the queue have the same architecture).
OpSys	the operating system of the machine or of the machines associated to the queue (assuming that all these machines run the same operating system).
PhysicalMemory	Minimum available physical memory (in bytes) associated to the queue.
LocalDisk	Local disk footprint
TotalCPUs	the number of total CPUs associated to the resource.
FreeCPUs	the total number of free processors associated to the resource, processors able to run, in that moment, jobs submitted to the resource.
NumSMPs	number of SMP processors associated to the resource.
TotalJobs	the number of jobs submitted to the resource, jobs that have not already been completed.
RunningJobs	The number of jobs submitted to the resource, that are currently running.
IdleJobs	the number of jobs submitted to the resource, jobs that are not running since they are waiting for available resources.
MaxTotalJobs	the maximum number of jobs (running and idle) allowed for the resource.
MaxRunningJobs	the maximum number of running jobs allowed for the resource.



Attribute	Meaning
WorstTraversalTime	Worst traversal time
EstimatedTraversalTime	Scaled value of the last traversal time
Status	the status of the resource. For a queue if it is ready or not to dispatch jobs to the executing machines.
RunWindows	the time windows that define when the resource is active, (for a queue: ready to dispatch jobs to the executing machines).
Priority	the priority of the resource.
MaximumCPUTime	the maximum CPU time allowed for jobs submitted to the resource.
MaximumWallClockTime	the maximum wall clock time allowed for jobs submitted to the resource.
MinSI 00	Min SpecI nt2000
MaxSI 00	Max SpecI nt2000
AvgSI 00	Average SpecI nt2000
RunTimeEnvironments	List of "tag(s)" identifying the appropriate test-bed SW installation (e.g. CMSVersion+CMSRoot or a list of RPM names/hashes)
AFSAvailable	Boolean indicating if AFS is available at the CE
OutboundIP	Boolean indicating that the resource has access to the internet
InboundIP	Boolean indicating that the resource can be accessed from the internet
StorageElements	List of SE URI's that are close "enough" to the resource (e.g. on the same LAN)

Table 5 classad Resource specific attributes