The Globus Resource Specification Language RSL v1.0

This is a document to specify the existing RSL v1.0 implementation and interfaces, as they are provided in the Globus v1.0 release. This document serves as a reference, and more introductory text.

The Globus Resource Specification Language (RSL) provides a common interchange language to describe resources. The various components of the Globus Resource Management architecture manipulate RSL strings to perform their management functions in cooperation with the other components in the system. The RSL provides the skeletal syntax used to compose complicated resource descriptions, and the various resource management components introduce specific <*attribute*, *value*> pairings into this common structure. Each attribute in a resource description serves as a parameter to control the behavior of one or more components in the resource management system.

This document is divided into the following sections:

RSL Syntax Overview

The core syntax of the RSL syntax is the *relation*. Relations associate an attribute name with a value, eg the relation ``executable=a.out" provides the name of an executable in a resource request. There are two generative syntactic structures in the RSL that are used to build more complicated resource descriptions out of the basic relations: *compound requests* and *value sequences*. In addition, the RSL syntax includes a facility to both introduce and dereference string *variables*.

The simplest form of compound request, utilized by all resource management components, is the conjunct-request. The conjuct-request expresses a conjunction of simple relations or compound requests (like a boolean AND). The most common conjunct-request in Globus RSL strings is the combination of multiple relations such as executable name, node count, executable arguments, and output files for a basic GRAM job request. Similarly, the core RSL syntax includes a disjunct-request form to represent disjunctive relations (like a boolean OR). Currently, however, no resource management component utilizes the disjunct-request form.

The last form of compound request is the multi-request. Multi-requests are used by DUROC, the coallocation component of the resource management system, to specify multiple parallel resources that make up a resource description. The multi-request form differs from the conjunction and disjunction in two ways: multi-requests introduce new variable scope, meaning variables defined in one clause of a multi-request are not visible to the other clauses, and multi-requests introduce a non-reducible hierarchy to the resource description. Whereas relations within a conjunct-request can be thought of as ``constraints'' on the resource being described, the subclauses of a multi-request are best thought of as individual resource descriptions that together constitute an abstract resource collection; the same attributes may be ``constrained'' in different ways in each subclause without causing a logical contradiction. An example of a contradiction would be to constrain the ``executable'' attribute to be two conflicting values within a conjunction.

The simplest form of value in the RSL syntax is the string literal. When explicitly quoted, literals can contain any character, and many common literals that don't contain special characters can appear without quotes. Values can also be variable references, in which case the variable reference is in essence **replaced** with the string value defined for that variable. RSL descriptions can also express string-concatenation of values, especially useful to construct long strings out of several variable references. String concatenation is supported with both an explicit concatenation operator and implicit concatenation for many idiomatic constructions involving variable references and literals.

In addition to the simple value forms given above, the RSL syntax includes the value sequence to express ordered sets of values. The value sequence syntax is used primarily for defining variables and for providing the argument list for a program.

RSL Tokenization Overview

Each RSL string consists of a sequence of RSL tokens, whitespace, and comments. The RSL tokens are either special syntax or regular unquoted literals, where special syntax contains one or more of the following listed special characters and unquoted literals are made of sequences of characters excluding the special characters.

The complete set of special characters that cannot appear as part of an unquoted literal is: `+' (plus), `&' (ampersand), `|' (pipe), `(' (left paren), `)' (right paren), `=' (equal), `<' (left angle), `>' (right angle), `!' (exclamation), `"' (double quote), `'' (apostrophe), `^' (carat), `#' (pound), and `$' (dollar). These characters can only be used for the special syntactic forms described in the above RSL syntax overview and in the RSL grammar provided below, or as within quoted literals.

Quoted literals are introduced with the `"' (double quote) or `'' (single quote/apostrophe) and consist of all the characters up to (but not including) the next solo double or single quote, respectively. To escape a quote character within a quoted literal, the appearance of the quote character twice in a row is converted to a single instance of the character and the literal continues until the next solo quote character. For any quoted literal, there is only one possible escape sequence, eg within a literal delimited by the single quote character only the single quote character uses the escape notation and the double quote character can appear without escape.

Quoted literals can also be introduced with an alternate *user delimiter* notation. User delimited literals are introduced with the `^' (carat) character followed immediately by a user-provided delimiter; the literal consists of all the characters after the user's delimiter up to (but not including) the next solo instance of the delimiter. The delimiter itself may be escaped within the literal by providing two instances in a row, just as the regular quote delimiters are escaped in regular quoted literals.

RSL string comments use a notation similar to comments in the C programming language. Comments are introduced by the prefix `(*'. Comments continue to the first terminating suffix `*)' and cannot be nested. Comments are stripped from the RSL string during processing and are syntactically equivalent to whitespace.

RSL Substitution Semantics

RSL strings can introduce and reference string variables. String substitution variables are defined in a special relation using the "rsl_substitution" attribute, and the definitions affect variable references made in the same conjunct-request (or disjunct-request), as well as references made within any multi-request nested inside one of the clauses of the conjunction (or disjunction). Each multi-request introduces a new variable scope for each subrequest, and variable definitions do not escape the closest enclosing scope.

Within any given scope, variable definitions are processed left-to-right in the resource description. Outermost scopes are processed before inner scopes, and the definitions in inner scopes augment the inherited definitions with new and/or updated variable definitions.

Variable definitions and variable references are processed in a single pass, with each definition updating the *environment* prior to processing the next definition. The value provided in a variable definition may include a reference to a previously-defined variable. References to variables that are not yet provided with definitions in the standard RSL variable processing order are replaced with an empty literal string.

RSL Attribute Summary

The RSL syntax is extensible because it defines structure without too many keywords. Each Globus resource management component introduces additional attributes to the set recognized by RSL-aware components, so it is difficult to provide a complete listing of attributes which might appear in a resource description. Resource management components are designed to utilize attributes they recognize and pass unrecongnized relations through unchanged. This allows powerful compositions of different resource management functions.

The following listing summarizes the attribute names utilized by existing resource management components in the standard Globus release. Please see the individual component documentation for discussion of the attribute semantics.

Common RSL attributes

rsl_substitution

GRAM attributes

arguments
count
directory
executable
environment
jobType
maxTime
maxWallTime
maxCpuTime
gramMyjob
stdin
stdout
stderr
queue
project
dryRun
maxMemory
minMemory
hostCount

DUROC Attributes

label
resourceManagerContact
subjobCommsType
subjobStartType

Simple RSL Examples

The following are some simple example RSL strings to illustrate idiomatic usage with existing tools and to make concrete some of the more interesting cases of tokenization, concatenation, and variable semantics. These are meant to illustrate the use of the RSL notation without much regard for the specific details of a particular resource management component.

Typical GRAM resource descriptions contain at least a few relations in a conjunction:

```
(* this is a comment *)
& (executable = a.out (* <-- that is an unquoted literal *))
    (directory  = /home/nobody )
    (arguments  = arg1 "arg 2")
    (count = 1)
```

Substitutions can be used to make sure the same substring is used multiple times in a resource description:

```
& (rsl_substitution  = (TOPDIR  "/home/nobody")
                       (DATADIR $(TOPDIR)"/data")
                       (EXECDIR $(TOPDIR)/bin) )
    (executable = $(EXECDIR)/a.out
            (* ^-- implicit concatenation *))
    (directory  = $(TOPDIR) )
    (arguments  = $(DATADIR)/file1
            (* ^-- implicit concatenation *)
                       $(DATADIR) # /file2
            (* ^-- explicit concatenation *)
                       '$(FOO)'          (* <-- a quoted literal *))
    (environment = (DATADIR $(DATADIR)))
    (count = 1)
```

Performing all variable substitution and removing comments yields an equivalent RSL string:

```
& (rsl_substitution  = (TOPDIR  "/home/nobody")
                       (DATADIR "/home/nobody/data")
                       (EXECDIR "/home/nobody/bin") )
    (executable = "/home/nobody/bin/a.out" )
    (directory  = "/home/nobody" )
    (arguments  = "/home/nobody/data/file1"
                       "/home/nobody/data/file2"
                       "$(FOO)" )
    (environment = (DATADIR "/home/nobody/data"))
    (count = 1)
```

Note in the above variable-substitution example, the variable substitution definitions are not automatically made a part of the job's environment. And explicit "environment" attribute must be used to add environment variables for the job. Also note that the third value in the arguments clause is not a variable reference but only quoted literal that happens to contain one of the special characters.

RSL grammar and tokenization rules

The following is a modified BNF grammar for the Resource Specification Language. Terminals appear in single quotes, eg `` `terminal' ``. Lexical rules are provided for the implicit concatenation sequences in the form of conventional regular expressions; for the ``implicit-concat" non-terminal rules, whitespace is not allowed between juxtaposed non-terminals. Grammar comments are provided in square brackets in a column to the right of the productions, eg [comment] to help relate productions in the grammar to the terminology used in the above discussion.

Regular expressions are provided for the terminal class ``string-literal'' and for RSL comments. These regular expression make use of a common inverted character-class notation, as popularized by the various `lex` tools. Comments are syntactically equivalent to whitespace and can only appear where the comment prefix cannot be mistaken for the trailing part of a multi-character unquoted literal.

```
specification                    [RSL String]
        => relation
        => '+' spec-list         [multi-request]
        => '&' spec-list         [conjunct-request]
        => '|' spec-list         [disjunct-request]
```

spec-list

```
        => '(' specification ')' spec-list
        => '(' specification ')'
```

relation

```
        => 'variables' = binding sequence        [variable def'ns]
        => attribute op value-sequence           [relation]
```

binding-sequence

```
        => binding binding-sequence
        => binding
```

binding

```
        => '(' string-literal simple-value ')'   [variable def'n]
```

attribute

```
        => string-literal                [attribute]
```

op

```
        => '=' |  '!=' | '>' | '>=' | '<' | '<='
```

value-sequence

```
        => value value-sequence
        => value
```

value

=> '(' value-sequence ')'
=> simple-value

simple-value

=> string-literal
=> simple-value '#' simple-value [concatenation]
=> implicit-concat
=> variable-reference

variable reference

=> '$(' string-literal ')' [variable ref.]
=> '$(' string-literal simple-value ')' [ref. w/ default]

implicit-concat

=> (unquoted-literal)? (implicit-concat-core)+ [implicit concat.]

implicit-concat-core

=> variable-reference
=> (variable-reference)(unquoted-literal)

string-literal

=> quoted-literal
=> unquoted-literal

quoted-literal

=> ' ' ' (([^'])|(' '' '))* ' ' '
=> ' '' ' (([^"])|(' '"' '))* ' '' '
=> '^' **c**(([^**c**])|(**cc**))* **c** [user delimiter]

unquoted-literal

=> ([^ \t\v\n+&|()=<>!"'^#$])+ [nonspecial chars]

comment

==> '(*' (([^*])|('*'[^)]))* '*)' [comment]