Contents

Chapter 1

Object Class Definitions

1.1 Software Objects

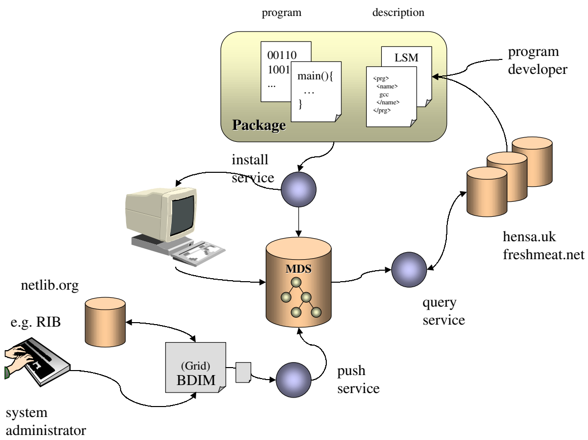

Figure 1.1: A suggested scenario of a Grid software service

The grid object model facilitates the building of systems which can make use of dynamically discovered software resources. Currently the following scenarios are considered:

1. publishing installed programs or software packages on a Grid compute resource.

2. publishing programs developed by Grid users under a user maintained directory sub tree.

3. dynamically updating of the information

4. support for the publication of software repositories in a tree structure

5. support for existing repository integration (e.g. Repository in a Box (RIB)

The following scenario demonstrates how such objects can be used in a Grid aware software service.

Scenario In a distributed computing environment, such as the Grid, one finds a large number of heterogenous computing resources. Such resources typically distinguish themselves not only by use of different hardware components, but also by the different software installed on such systems.

Today an application user is force, to make the decision to locate an appropriate compute resource which provides the necessary software infrastructure to run the application. This is typically determined by hand, while obtaining an account on the machine and inspecting the available infrastructure.

Publication of the information related to installed software on the system provides the advantage that an application user is able to dynamically select from a potentially very large number of computing resources, the one which has, for example,

- all necessary software components installed
- a machine on which all the necessary components to run the application can be installed
- in combination with performance data, a machine on which the application is executed in minimal wallclock time.

Hence, it is desirable to provide a service, which allows the tracking of software at installation time, as well as, runtime. We recommend to base such services on the objects described in this section.

We feel that the objects defined in this section provide a minimal common denominator to support a large class of such services.

The representation of software is based on three main object classes. Two of the classes that deal with software installed on a compute resource are

- Grid::Software and
- Grid::Daemons.

A third object class,

- Grid::SoftwareRepository

supports the definition of a tree based representation for the classification of software. This allows the support for software repositories such as the "Alliance software repository."

To support the host centric DIT structure of the Grid object model, the Grid::Software and Grid::Daemon objects are stored under the hosts on which the software is installed or running. We recommend placing the Grid::SoftwareRepository objects also under the hosts on which the software repository is installed.

Next we describe the functionality of the objects in more detail.

Grid::Software This object class is used to represent programs and program packages installed on a ComputeResource. Examples for software objects are packages (e.g., PETSc) and utility programs (e.g., compilers). Since instances of these objects are typically accessed through a particular compute resource, they are placed in the DIT underneath a compute resource. In case they are stored in a shared filespace we currently recommend simply replicating the information, if desired, or make a pointer to an object which describes the software for a group of machines.

Grid::Daemon This object is used to represent programs that run on a compute resource that provides a particular service. Examples of such services include database servers and programs needed in order to run Globus on a compute resource (e.g., hbm, gloperf).

Grid::SoftwareRepository A Grid::SoftwareRepository object is a customized object used to store the information associated with a software repository. This object is mainly used to define a hierarchical ordering on software packages in order to classify them (e.g., an Alliance hierarchy tree).

The rest of this chapter explicitly specifies the objects and their attributes. The chapter concludes with some simple examples for using the objects.

1.1.1 Grid::SoftwareVersion

Each piece of software or software package has been released at a particular time and includes in most cases a well known string or tag describing the release. In many cases this release string can be broken down in many convenient attributes. In the case of Grid related software, we recommend filling out the release information below as much as possible. This will allow one to easily design sophisticated queries without the parsing of the releaseString while reducing ambiguity and complicated release tags. Grid::SoftwareVersion

```
Grid::SoftwareVersion OBJECT CLASS
  MUST CONTAIN {
    releaseTag          :: cis, singular,
  }
  MAY CONTAIN {
    releaseDate         :: cis, singular,
    releaseMajor        :: cis, singular,
    releaseMinor        :: cis, singular,
    releasePatch        :: cis, singular,
    releaseBeta         :: cis, singular,
 }
```

Attributes

releaseTag:	(cis) the tag usually coming with the release
releaseDate:	(cis) the data and time of the release
releaseMajor:	(cis) the major release number
releaseMinor:	(cis) the minor release number
releaseBeta:	(cis) the beta version
releasePatch:	(cis) the patch version

1.1.2 Grid::SoftwareRepository

A RepositorySoftware object stores information related to software repositories [?]. Each repository object contains the Grid::SoftwareVersion. Grid::SoftwareRepository

```
Grid::RepositorySoftware OBJECT CLASS
  SUBCLASS OF Grid::Top
  RDN = rsw (repositorySoftware)
  CHILD OF {
     Grid::organization;
     Grid::organizationalUnit;
     Grid::ComputeResource;
     Grid::Root;
```

```
}
MUST CONTAIN  {
   name                     :: cis, singular,
   description              :: cis, singular,
   dateOfInformation        :: cis, singular,
   deploymentStatus         :: cis, singular,
   domainClassification     :: cis,
   keywordClassification    :: cis,
   targetEnvironment        :: cis,
   documentation            :: cis,
   contact                  :: dn,
   downloadURN              :: cis,
}
```

Attributes

name: (cis) the name of the software package, e.g. MPICH

description: (cis) a small description, e.g. MPICH is a freely available, portable implementation of MPI, the new Standard for message-passing libraries.

dateOfInformation: (cis) the date when the software was installed, e.g. Tue Jul 14 13:37:22 1998

deploymentStatus: (cis) a URL which describes the status of the installation e.g. [?]

domainClassification: (cis) a classification of the domain the software falls into e.g. Parallel Processing Tools!Communication Libraries

keywordClassification: (cis) a keyword to enable a simple classification e.g. message passing library

targetEnvironment: (cis) the target environment can describe constraints under which the program can be executed. The string

(&(os=solaris)(cc=gcc)(architecture=sun5))

indicates that the target environment is a sun5 machine running solaris and where the gcc compiler is available. The values specified should conform to the values provided by grid services.

documentation: (cis) pointers to the documentation, e.g. http://www.mcs.anl.gov/mpi/mpich/

contact: (dn) distinguished name of the point of contact for the software

downloadURN: (cis) information about urls and ftp servers for a direct download the mpich full distribution file

1.1.3 Grid::Software

Grid::Software

A GridSoftware object stores information related to installed software or software packages. The software object is also used as part of a software repository to allow a convenient classification tree.

```
Grid::Software OBJECT CLASS
  SUBCLASS OF Grid::Top
  RDN = sw (software)
  CHILD OF {
     Grid::organization;
     Grid::organizationalUnit;
     Grid::ComputeResource;
     Grid::ResourceManager;
     Grid::ResourceManagerQueue;
     Grid::RepositorySoftware;
     Grid::Root
  }
  AGGREGATES {
        Grid::Version;
  }
  MUST CONTAIN {
    name                 :: cis, singular,
    description          :: cis, singular,
    documentation        :: cis,
  }
  MAY CONTAIN  {
    executableDirectory :: ces,
    manpathDirectory    :: ces,
    manualEntry         :: cis,
    libDirectory        :: ces,
    includeDirectory    :: ces,
    rootDirectory       :: ces,
    sharedlibraryDirectory :: ces,
    bugs                :: cis,
    dateOfInstalation   :: date, singular
    installedBy         :: dn,
    administeredBy      :: dn,
    notes               :: cis,
    repositoryPackage   :: dn,
    configuration       :: ces,
    installPrefix       :: ces,
    state               :: cis,
    environment         :: ces,
    keyword             :: cis,
    interface           :: ces,
    quality             :: cis,
  }
```

Attributes

name:	(cis) a name or abbreviation for the software
quality:	(cis) a quality value which can be attached to the installation of the software to indicate the quality of the installation. We suggest using the following keys: production, experimental, and temporary
dateOfInstalation:	(dis) The date when the last installation has been performed.
manualEntry:	(cis) usually the name of the program or a package.

description: (cis) describes what the software is doing or gives an easy to understand identifier e.g. GNU CC compiler

documentation: (cis) gives pointers where one can find documentation on the software e.g. http:/www.gnu.org/gcc man gcc

executableDirectory: (ces) specifies the directory in which the executable can be found e.g. /usr/local/bin (directories could also be specified as a URL)

manpathDirectory: (ces) specifies the directory in which the man pages can be found, e.g. /usr/local/man

libDirectory: (ces) specifies the directory in which the libraries can be found, e.g. /usr/local/lib

rootDirectory: (ces) specifies the root directory (if present) of the software package

includeDirectory: (ces) specifies the directory in which the include files are located, e.g. /usr/local/include

sharedlibraryDirectory: (ces) specifies the directory in which the shared libraries can be found

bugs: (cis) specifies links to known bugs

installedBy: (dn) specifies the dn of the person who installed the software

administeredBy: (dn) specifies the dn of the person who administers the software

notes: (cis) a field which can hold notes and remarks

repositoryPackage: (dn) a distinguished name or string of a repository package this software which this softwrae could belong to.

configuration: (cis) parameters given at configuration time

installPrefix: (ces) starting location of libraries, header files, etc.

state: (cis) a state which might describe the a state associated with the object e.g. active,

environment: (ces) this field is used for special software which relies on setting special environment conditions

keyword: (cis) this field contains a keyword which might make searching easier. It is used for Classification purposes.

interface: (cis) this field can be used to describe input and output parameters of software objects. This field is experimental and might be removed.

1.1.4 Grid::Daemon

Grid::Daemon

A Grid::Daemon stores information related to daemons running on a system Examples for a daemon are gatekeeper, HBM datacollector, ...

```
Grid::Daemon OBJECT CLASS
  SUBCLASS OF Grid::Software
  RDN = da (daemon)
  CHILD OF {
```

```
    Grid::ComputeResource;
    Grid::Root
  }
  AGGREGATES {
    Grid::DaemonAccess;
  }
  MAY CONTAIN  {
    hostname            :: cis, singular,
    hostnameDN          :: dn,  singular,
    name                :: cis, singular,
    parameter           :: cis,
    developerContact    :: dn,
    daemonAdministrator :: dn,
  }
```

Attributes

name: (cis) specifies what it's function or type of daemon it is, e.g. HBM datacollector

daemonAdministrator: (dn) specifies an administrator for bug notification

developerContact: (dn) pointer to the dn that describes how to contact the developers.

parameter: (cis) specifies a tuple of the form (attribute=value) in order to set specific values necessary to configure the daemon. This is only important if schema checking is set to true. If schema checking is set to false the attributes and values can be stored directly into the object. Using the parameter attribute is preferred. E.g. parameter: updateInterval 10

1.1.5 Grid::DaemonAccess

Grid::DaemonAccess

The object Grid::DamonAccess is an aggregate object, that is not part of every daemon.

```
Grid::DaemonAccess OBJECT CLASS
  MAY CONTAIN{
        accessProtocoll ::cis ::singular,
        accessPort      ::cis ::singular,
        controlPort     ::cis ::singular,
        controlProtocol ::cis ::singular,
  }
```

Attributes

accessProtocol: (cis) name of communications daemon (telnet)

accessPort: (int) access port number

controlProtocol: (cis) control/data protocol

controlPort: (cis) the control port number

1.1.6 Example Software Object Class Definitions

GridSoftware

This example illustrates how to store information about an installed program and an installed software package on a computer. To keep the information about the installed software separated from the other information available on the machine, we recommend storing all software related objects in the subtree gn=software.

Thus a typical DIT would look like:

```
o=grid,
   ou=Argonne National Laboratory
      ou=Mathematics and Computer Science Division
         hn=dark.mcs.anl.gov
            gn=software
               sw=Globus
                  ...
                  releaseTag: 1.0
               sw=hbm
                     ...
               sw=gcc
                     ...
               sw=LAPACK
                     ...
               sw=PETSC
                     ...
```

The Globus software is currently stored directly underneath the host to ensure compatibility to earlier versions of Globus. We expect that the software object will soon be moved under gn=software.

Example:
Grid::software

```
dn: sw=Globus, hn=pitcairn.mcs.anl.gov, ou=Mathematics and
 Computer Science Division, o=Argonne National Laboratory, o=Grid
name: Globus
description: The Globus toolkit provides the infrastructure to build
 metacomputing applications
documentation: http://www.globus.org/documentation
executableDirectory: /soft/pub/bin
manpathDirectory: /soft/pub/man
libDirectory: /soft/pub/lib
includeDirectory: /soft/pub/include
bugs: bugs@globus.org
installedBy:cn=Gregor von Laszewski, ou=Mathematics and Computer
 Science Division, o=Argonne National Laboratory, o=Grid
administeredBy: cn=Joe Bester, ou=Mathematics and Computer Science
 Division, o=Argonne National Laboratory, o=Grid
repositoryPackage: Globus
releaseTag: 0.9.9.a
releaseDate: Fri Aug 28 14:02:08 GMT 1998
releaseMajor: 0.9
releaseMinor: 9
releasePatch: a
configuration: --enable-hbm -enable-gloperf
installPrefix: /soft/pub
```

GridDaemon

The following example shows the entry of an LDAP server daemon. Other programs running on the system could be described similarly.

```
o=grid,
   ou=Argonne National Laboratory
      ou=Mathematics and Computer Science Division
         hn=dark.mcs.anl.gov
            gn=software
               ...
            da=MDS
```

Example: GridDaemon

```
dn: da=MDS, hn=dark.mcs.anl.gov, ou=Mathematics and Computer
 Science Division, o=Argonne National Laboratory, o=Grid
accessProtocoll: LDAP
accessPort: 389
controlPort: http
controlProtocol: 13555
name: Local Argonne MDS
daemonAdministrator: cn=Gregor von Laszewski,
 ou=Mathematics and Computer Science Division,
 o=Argonne National Laboratory, o=Grid
developerContact: g=Globus, g=projects, o=Grid
parameter: backupintervall 3600
```

GridSoftwareRepository

An example for a software repository can be found at the NHSE. Software repositories must have a structure in which software is classified. It is important to note that the Grid based DIT automatically provides a repository based on the organizations.

In case one likes to just maintain a subset based on other classification criteria, the repository object can be used. The object definition for the Repository is flexible enough to allow user defined classification repositories.

In the following example, we create a repository which represents message passing libraries.

```
o=Grid
  o=University of Illinois at Urbana
    ou=National Computation  Science Alliance
      ou=Alliance Software Repository
          rsw=Parallel Processing Tools
             rsw=Communications Libraries
                rsw=MPICH
            ...
                rsw=PVM

                rsw=MPL
```

Example: GridSoftware-Repository

```
dn: rsw=MPICH, rsw=Parallel Processing Tools, rsw=Communications
 Libraries, ou=NPACI Software Repository, ou=National Computationl
```

```
 Science Alliance, o=University of Illinois at Urbana, o=Grid
name: MPICH
description: MPICH is a freely available, portable implementation of
 MPI, a standard for message-passing libraries.
dateOfInformation: Tue Jul 14 13:37:22 1998 GMT
domainClassification: Parallel Processing Tools!Communication Libraries
 keywordClassification: message passing library
targetEnvironment: (|(\&(os=solaris)(cc=gcc)(architecture=sun5))
  ( \&(os=irix)(cc=cc)(architecture=mips)))
releaseTag: 1.1.1
releaseDate: Sat Jun 12 00:45:20 UTC 1999
documentation: http://www.mcs.anl.gov/mpi/mpich/
contact: cn=Rusty Lusk, ou=Mathematics and Computer Science Division,
o=Argonne National Laboratory, o=Grid
downloadURL: ftp://ftp.mcs.anl.gov/pub/mpich/mpich.tar.gz
```

1.2 Repository in a Box Interface for the Grid

The Repository in a Box (RIB) strives to provide a simple data model and tools to manipulate and handle repository information and publish the information on a web server.

RIB is being used by the Alliance to describe the information about a subset of Grid software deployed at the Alliance sites[]. The current usage model for this approach assumes that the system administrator maintains the list of installed software components and machine descriptions by hand.

Though this model is sufficient for the Alliance repositories it is important to consider that the Grid is highly dynamic and a large amount of software will be installed on the existing hardware infrastructure. Thus, the more the information update process can be automated, the easier the maintenance of the information will be. A good model for such a system is suggested by the Globus project. Here the combination of automatic information recovery and the usage of RIB to create nice looking WWW pages can lead to a product which may be ultimately needed by the larger Grid community to display installed software components.

The design of this model is based on the fact that information can be either obtained from the current RIB database, the Grid MDS, or other repositories. This data will then be transferred into a uniform format and stored in a repository. It would be trivial to generate two different formats one based on the MDS object definitions in order to update the MDS and one based on IEEE BIDM which is then used by RIB to produce the web pages.

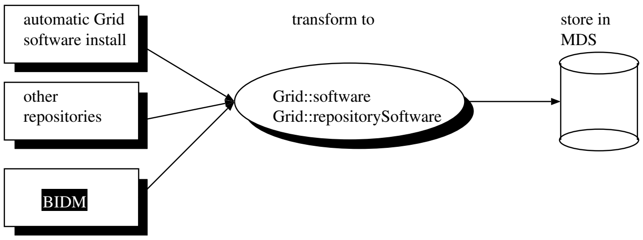

Figure 1.2: A suggested scenario of a Grid software service

1.2.1 Possible usage Scenario

We suggest that each software package which is developed as part of the Alliance, PACI, and DOE2000 projects must include a simple standard description about the package in the object description.

In case the developer is part of the Alliance, DOE2000, or IPG efforts we suggest that the addition of a standard software description page is part of the project deliverable. The developer gets reworded with a Grid compliance certificate.

The definition of such a description can be done by the developers of each project individually and will not pose a large overhead. Instead of defining such objects in LDAP, we suggest using XML notation since it is possible to check it against a document type definition, as thus minimizing errors based on syntactic or incomplete data. The design of such an XML object follows exactly the LDAP notation (as defined elsewhere).

Once a project/package is in publishable stage, a registration can be performed either with a registration program or via a Web entry format.

Users or sites which want to use and install this "grid compliant" software automatically get with the package a program that populates the software information to the appropriate repository as part of the MDS.

1.2.2 Mapping the RIB Object Model to the Grid Object Model

The RIB technology uses three main objects. These objects are as follows:

Asset An asset describes an entity, e.g. a software package which is to be deployed on a machine.

Machine A machine object describes a machine on which Assets my be deployed.

Deployment A deployment object contains information arising at deployment time about a particular asset on a machine. Such information includes, for example, when and who has deployed the asset.

In addition to the objects a relationship in RIB between the objects is expressed with pointers. In the following section we annotate attributes with (*RIBpointer*) in order to express that these attributes in the original RIB model are used to point to another object instead of carrying string based information.

1.2.3 Asset

The following list describes the functionality of each attribute in the RIB object and contains recommendations on how to modify them in order to utilize the Grid::software and Grid::softwareRepository objects.

We have currently not integrated the attributes into Grid::software and Grid::repositorySoftware in order do allow discussion of this model with a larger audience.

<u>abstract:</u> substitute — (... lost description ...)

Add (cis) Grid::software:shortDescription since the term abstract is too ambiguous.

<u>cost:</u> (cis) (... lost description ...)

We propose to define a grid working-group to research cost related objects. Temporarily, we suggest to add Grid::software:cost.

dateOfInformation: (cisdate) Unchanged Grid:software:dateOfInformation or Grid:softwareRepository.

distributionStatement: (cis) (... lost description ...) We propose to form a working-group researching this issue in more detail. Temporarily we suggest to add Grid::software:distributionStatement and Grid::repositorySoftware:distributionStatement.

domain: substitute — We suggest to use the keyword attribute and add the domain simply as keyword in order to reduce the attributes. Additionally we suggest to use the (cis) Grid::softwareRepository:domainClassifica

icon: (binary) We propose to add Grid::softwareRepository:icon.

keyword: (cis) Use the Grid::software:keyword and Grid::softwareRepository:keyword.

language: (cis) Programming language used to express one or more elements of the asset, we suggest to add add Grid::software:programmigLanguage. We find language to ambiguous.

name: (cis) Same as in Grid::software and Grid::repository software.

restrictions: (cis) We suggest to build a separate working group to research this issue. Temporarily we suggest to add

Grid::software:restrictions and Grid::repositorySoftware:restrictions.

targetPlatform: remove — Computer system, operating system, and/or compiler for which the asset was developed. The RIB concept of target platform and targetSoftware is too limited to be useful for the Grid. Thus, we recommend to describe the properties as a target environment attribute

Grid::softwareRepository:targetEnvironment.

Values describing the target environment could be stored in tuples like:

(&(os=solaris)(cc=gcc)(machine=sun4))

targetSoftware: remove — See TargetPlatform.

targetEnvironment: See TargetPlatform. A better way to describe targetSoftware and targetPlatform.

titleLine: (cis) A one-line description of the asset. add Grid::softwareRopository:oneLineDescription. The term titleLine assumes a usage of the attribute "oneLineDescription", thus the attribute should be really called "oneLineDescription".

version: (object) Version designation of the asset use the Grid::version aggregate object.

versionDate: (cisdate) Publication date for the given version. use the Grid::software:releaseDate or Gridd::repositorySoftware:releaseDate.

webpage: (cis) A web page URL associated with the asset. Add Grid::softwareRepository:webpage. This points to the home page of the software, this could also be part of the description.

contactIs: (RIB pointer) Person or organization acting as point-of-contact for the asset.

This should be replaced by a dn pointer to the person or group that administers the original software. If a person does not exist, it should be added to the appropriate institution.

(dn) Grid::softwareRepository::administrator

In case the software is installed on a computer, an administrator can be added to the software.

(dn) Grid::software::administrator.

Here the administrator would refer to the person at the local site who installed the software and should be contacted in case local install questions occur.

specialContact: (RIB pointer) A word or phrase explaining special contact circumstances. The Grid team has no practical experience with this attribute. Temporarily, we suggest to add

Grid::softwareRepository:specialContact.

awardedWith: (RIB pointer) Pointer to a certificate which the asset has been awarded. The Grid team has no practical experience with this attribute. It Temporarily we suggest to add

Grid::softwareRepository:awardedWith.

isComposedOf: (RIB pointer) Indicates an asset that is a subset of the current asset.

In many cases the DIT structure defines substitutes the IsCOmposedOf. In the case where a software object is below another, it means that it is a part of a larger package.

Nevertheless, it should be possible to place pointers to minimize duplication. The pointers will point to the repository.

Add Grid::software:isComposedOf and Grid::repositoySoftware:isComposedOf.

isLocatedIn: (RIB pointer) Indicates a library (repository) where the asset is located. This is the reverse of IsLocatedIn

add Grid::software:isComposedOf and Grid::repositoySoftware:isComposedOf.

isMadeOf: (RIB pointer) Indicates elements (files) that are part of the asset.

One of the few attributes, which we find too difficult to maintain. This field should be discovered automatically.

wasCreatedBy: (RIB pointer) One of the few attributes that we do not see to be necessary.

1.2.4 Deployment

The list displays the deployment object as found in RIB. Depicted are the attributes and the description of the attributes.

contactIs: (RIB pointer) Person or organization responsible for deploying the software.

deploymentDate: Date the software was deployed.

deployedOn: (RIB pointer) Machine on which the software is deployed.

isDeploymentOf: (RIB pointer) Asset being deployed.

status: Status of deployment (e.g. pending, complete).

usageInfo: A URL providing information for this particular deployment.

Much of this deployment information is implicitly defined by the position of the Software/Asset in the DIT.

Thus, the deployedOn can be derived from the distinguished name of the deployed asset.

For example, "sw=gcc, g=software, hn=speedy.anl.gov, o=grid" indicates that the gcc compiler is installed on the host speedy.anl.gov. If desired, the sw object can contain a dn pointer pointing to a repository describing the gcc compiler. The "isDeploymentOf" is substituted by a dn pointer. The usageInfo is in the Grid model named "description". The "deploymentDate" is the same in both models. The "contactIs" replaced by a distingushed name pointer to the object which is associated with the person or organization responsible for deploying the software.

In summary,

contactIs (dn) (RIB pointer) replaced by dn Pointer to Grid::software:administeredBy for deploying the software.

deploymentDate (cisdate) This is supported by the Grid::software:deploymentDate attribute.

deployedOn (dn) (RIB pointer) replaced by the position in the DIT. In order to support shared software deployment on a shared filesystem, we recommend to add a dn pointer pointing the the machines or group of machines on which the software is installed.

(dn) Grid::software:deployedOn

isDeploymentOf (RIB pointer) replaced by a dn pointer to a repository object describing the Asset being deployed. This is implementation specific, e.g. what is the repository.

(dn) Grid::software:isDeploymentOf

status removed — Operationally, it is sufficient to only accept "installed" and "not installed" as values. Thus, a software object in the DIT would indicate whether the software installation is completed, thus making the status field unnecessary.

usageInfo replaced — same as (cis) Grid::software:description and/or (cis) Grid::softwareRepository:description.

1.2.5 Machine information in RIB and MDS

RIB defines the object of a machine on which a software package is deployed. This machine object is quite limited and is superseded in most every aspect by the GridComputeResource object. Thus instead of using the information stored in RIB we suggest to use the dynamically discovered information out of the MDS. Since by definition all machines will have Globus installed this is possible. In case the machine is not published, one has either to install the machine discovery scripts which are part of the Grid software (e.g. Globus) or input this information by hand. Again, the Globus team suggest to discover as much information as possible automatically. There should not be the need for the system administrator to even think about what the cache size of a machine is.

Furthermore, we suggest that the RIB team develops a simple shell script which transforms the information in LDIF format appropriate to the GlobusComputeResource objects.

Since most of the machine related information can be obtained form the Grid MDS, we feel it is not necessary to push any machine related information stored in RIB into the MDS. In contrast, we feel that some of the information provided by the MDS is more accurate than the information in RIB since it is not discovered by a user but by automatic program execution returning attributes in a standard format.

dateOfInformation:	dateOfInformation in "date -u" format.
processorsPerNode:	automatically discovered by Globus and/or the Globus model provides more accurate information.
manufacturer:	automatically discovered by Globus and/or the Globus model provides more accurate information.
clockSpeed:	automatically discovered by Globus and/or the Globus model provides more accurate information.
peakPerNode:	automatically discovered by Globus and/or the Globus model provides more accurate information.
memoryPerNode:	automatically discovered by Globus and/or the Globus model provides more accurate information.
cacheSize:	automatically discovered by Globus and/or the Globus model provides more accurate information.
operatingSystem:	automatically discovered by Globus and/or the Globus model provides more accurate information.
architecture:	not supported by RIB.
networkInterfaces:	not supported by RIB.
interconnectSpeed:	not supported by RIB.
webpage:	description, url.
BelongsTo:	(RIB pointer) not supported by RIB.