CHAPTER 4

SEMANTIC DESCRIPTION AND DISCOVERY OF GRID RESOURCES

4.1 GENERAL

The research work described in this thesis focuses on developing a semantic component in conventional Grid architecture to support ontology based representation of Grid resource metadata and to facilitate context based information retrieval that complements Grid schedulers for effective resource management. The previous chapter focussed on semantic representation and discovery of Grid services whereas this chapter considers semantic representation and discovery of computational resources participate in Grid. Earlier research works in semantic Grid has suggested a knowledge layer in conventional Grid architecture that supports semantic representation of Grid metadata and include components for knowledge retrieval from it. Similarly, the semantic component proposed in this work interacts with the underlying Grid resources and creates ontology representation of it to support semantic based information retrieval. Protégé is used for representation of computation resources metadata and Algernon inference engine is used for discovery of resources.

4.2 ARCHITECTURE

A four layered semantic Grid architecture is modelled with knowledge layer sitting at the top of Grid scheduler in the high level Grid middleware layer as shown in Figure 4.1. This architecture is inspired by four layered standard Grid architecture proposed by Foster et al (2002). The
proposed knowledge layer component can very well fit in high level collective layer proposed in standard Grid architecture as the components present in the knowledge layer aggregates metadata of all the participating computation resources and describe them using semantic approach which in turn helps in semantic discovery.

![Layered architecture supporting ontology based Grid resource description and discovery](image)

*Figure 4.1  A Layered architecture supporting ontology based Grid resource description and discovery*

**Fabric Layer**

The Grid fabric layer provides the resources to which shared access is mediated by Grid protocols. The resources may be computational resources, storage systems, catalogues, network resources and sensors or may be a
logical entity, such as a distributed file system, computer cluster, or distributed computer pool.

**Core Middleware Layer**

This layer consists of low-level middleware that provides secure and unified access to remote resources. Depending on the type of resources, we can choose different middleware such as Globus, Unicore, Alchemi, and Storage Resource Broker. Using services of such low-level middleware layer, one can create high-level middleware services that support rapid creation and deployment of applications on global Grids.

**High Level Middleware layer**

In our architecture, this layer is proposed to be implemented using a Grid scheduler. It discovers suitable resources that match the user requirements and schedules job to that selected resource. Further, the scheduler monitors the execution of the job in the resource and aggregates the results of the execution. It obtains the services provided by the knowledge layer for discovery of suitable resource.

**Knowledge layer**

The knowledge layer provides knowledge discovery from the data aggregated from the information services of the underlying middleware layer. Moreover, this layer is domain oriented and generally uses domain knowledge built with domain ontology. In the proposed architecture, the knowledge layer contains necessary functions and services for creating ontology description of the available Grid resource information. It also provides programming and user interfaces that enables semantic based discovery of suitable Grid resources using the ontology description maintained in this layer. The knowledge layer is modelled as a separate layer and is not tightly coupled
with the underlying middleware layer. In this work, it has been implemented in such a way that it creates ontology description of Grid resources present in a text file or from a database. Also, it can directly query the globus based Grid resources to aggregate the resource information and creates ontology description.

**Application layer**

The application layer enables the use of resources in a Grid environment through various collaboration and resource access protocols. The discovery portal present at this layer allows the resource requester to submit resource requirements to find out suitable resources for application execution. It also includes software and tools to support application workflow and composition.

**4.3 KNOWLEDGE LAYER FOR GRID RESOURCE DESCRIPTION AND DISCOVERY**

The components proposed in knowledge layer can be divided into two types based on their functionalities as resource description and discovery component as shown in Figure 4.2. The resource description module supports representation of Grid metadata whereas discovery module discovers suitable resources for application execution. Both the modules implemented in the knowledge layer use semantic web’s approach of making information understandable by computers with the help of ontology.

Based on Karlshruhe ontology model (Stumme and Enrig 2003), an ontology with datatypes can be defined as a structure $O:=(C, T, \leq_C, R, A, \sigma_R, \sigma_A, \leq_R, \leq_A, I, V, t_C, t_R, t_A)$ consisting of
Six disjoint sets C,T,R,A,I and V called concepts, datatypes, relations, attributes, instances and data values respectively.

Partial orders \( \leq_C \) on C called concept hierarchy or taxonomy and \( \leq_T \) on T called type hierarchy.

Functions \( \sigma_R: R \rightarrow C^2 \) called relation signature and \( \sigma_A: A \rightarrow C \times T \) called attribute signature.

Partial orders \( \leq_R \) on R called relation hierarchy and \( \leq_A \) on A called attribute hierarchy, respectively.

A function \( t_C: C \rightarrow 2^I \) called concept instantiation.

A function \( t_T: T \rightarrow 2^V \) called datatype instantiation.

A function \( t_R: R \rightarrow 2^{IxI} \) called relation instantiation.

A function \( t_A: A \rightarrow 2^{IxV} \) called attribute instantiation.

Also, we can define ontology \( O \) on a domain \( M \) as a triple \( O = (T_C, T_R, I_O) \) in which

\( T_C \) (respectively, \( T_R \)) is set of concept types (respectively, relation types) relating to domain \( M \).

\( I_O \) is a set of instances semantically related to the domain \( M \). An element of \( I \) is an instance or a realization of concept type in the real world.

We can also define other relations that exists between resources namely,

Subsumption relation

\[ \leq: (T_C \times T_C) \cup (T_R \times T_R) \rightarrow \{ \text{true, false} \} \] for example “\( \leq (c,d) = \text{true} \)” means: “\( c \) is a subtype of \( d \)”. 
- Conformity relation, which relates each individual instance in I to one, and one only, concept type in $T_C$.

The semantic component exploits the above said properties in creating ontology of Grid resource and suitable resource discovery.

### 4.3.1 Resource Description using Ontology template

Creating ontology is inevitably a very laborious process, and there is a need to at least partially automates the process of ontology creation and knowledge extraction. Hence, we can imagine a predefined ontology of concepts and relationships, plus a knowledge base of instances. To realize this, the resource description module defines resource ontology template created using Protégé editor and it provides necessary concepts and properties with which a resource can be described. Different possible computing resources are considered for creating ontology template.

![Figure 4.2 Semantic Component](image-url)
Our structuring of the ontology of resources is motivated by the need to provide semantic information about a resource and also to provide transparent access to Grid resources. We propose the following precise definitions to explain the motivation behind the creation of ontology template and how it can be used for semantic description.

**Definition 1:** An ontology template is a domain specific ontology that provides hierarchy of concepts along with properties to define their characteristics.

**Definition 2:** Any resource can be modelled as an instance of a specific concept provided that the resource can be described using the properties defined in that concept.

Once the ontology template is created, knowledge base can be built with the instances and the specific property instantiations. Together the ontology and the knowledge base make up a semantic repository. When a resource is registered in the Grid, its information can be obtained using Grid resource monitoring tool like MDS (MDS 2005) of Globus Middleware. With this information, instance of appropriate resource concept in the ontology template is created for every computing resource in the Grid. The characteristics of the resource for ex, freeRAMSpace, is also defined in the respective property of the appropriate resource concept in the ontology template. For example, an existence of a computer with Linux OS can be represented in the ontology template by creating an instance for the concept “computer” and the “hasOS” property of the concept “computer” will be assigned the value “Linux”. A portion of ontology template is shown in Figure 4.3.
Figure 4.3 Ontology template

Protégé-OWL APIs is used to dynamically create instance of a particular concept and also to assign values to appropriate properties in the resource ontology template. With these features, the resource information of the entire Grid environment can be described semantically which in turn enables semantic discovery of Grid resources.

4.3.2 Semantics based Resource Discovery

Grid schedulers obtain resource requirement from the users for application execution, performs matchmaking of available resources against
the requested ones, and discovers suitable resources that matches the requirements. In Grid, every participating resource advertises their capabilities in terms of properties such as Number of CPUs, Memory, storage capacity, CPU Load etc., through a Grid scheduler. Matchmaking refers to capability matching of the requested Grid resources with the advertised ones. Grid scheduler implements a mechanism to support expression of application requirements that includes nature of task, type of computing resources it requires, their configuration in terms of properties mentioned above in addition to other parameters such as budget, deadline and at what time the resource is actually needed. Some scheduler such as condor also allows to specify the resource requirements through mathematical expression (such as $<$, $>$, $<=$, $>=$, and $==$). For instance, a sample job submission request supported by condor is given below:

```plaintext
Executable = /bin/ls
Requirements = [Memory >= 128 && OpSys == "LINUX_2.4" && Arch =="i686"]
Rank = [Memory >= 256]
Error = ls.err.$(Process)
Output = ls.out.$(Process)
```

The matchmaking algorithm evaluates the requirements present in this request and finds out suitable resources available in the Grid by matching the keywords present in the request. Such keyword based matchmaking mechanism retrieves the resource that exactly matches the requirements. However, there are situations when potential resources available that can possibly execute the application but still miss from the search. Keyword based matchmaking mechanism is not able to find such resources and hence no suggestion can be made to the user about these resources. This is due to the fact that the algorithm does not understand the semantic relationship between the keywords. However, semantic relationship can be established by the ontology representation of Grid resources and their characteristics and
modifying matchmaking algorithm to exploit the same during resource discovery phase.

The resource discovery module proposed in this work implements an ontology based matchmaking mechanism that determines semantic relationship between the request and the advertised resource information and hence determines closely related Grid resources when exact match fails. The algorithm classifies resources into three broad categories, namely exact, resources that exactly match the requested resource requirements, subsume, if the advertised resources have more capabilities than that of requested, plugin, an exact contrary to the previous case, that is, the application expects more capabilities than that is advertised, and disjoint, which actually is not a match but infers that both the request and the available resources are completely different.

In Grid context, it is possible to consider the resources that fall into first two categories for application execution while the plugin resources cannot execute the application. This can be very well explained with the following scenario. Let us consider that an application requires Linux 2.4 version of Operating system and 5 numbers of CPUs for execution. An instance of Linux 2.4 or Linux 2.6 operating system with more than 5 CPUs falls into subsume categories whereas an instance of Linux 2.2 operating system with less than 5 numbers of CPUs would fall into plugin category. In this case, the plugin resource cannot execute the application as it expects more capabilities than that are available.

To determine semantic relationship, the matchmaking system has to interact with the ontology knowledge base and retrieve information from it. Inference engines are widely used for this purpose and in this research work, Algernon inference engine, developed by Stanford University is used for
information retrieval from the knowledge base. It offers versatile queries that can be executed onto the knowledge base. The discovery module based on the user’s request generates appropriate Algernon based query and executes it into knowledge base. If the exactly matching resources are not available, it then determines the resources that exhibit subsumption relation with the request. This kind of inference is possible if the requested concept is modelled as subconcept of the advertised concept in the ontology template, then it is possible to conclude that there exists a subsume match between the advertised and requested capabilities. For instance, while constructing computational resource ontology, a resource with Linux operating system would be modelled as subconcept of the Unix based resource. In such cases, it can be concluded that a request of resource with Unix operating system subsumes an advertisement of a resource with Linux operating system.

The resources identified from this module can be suggested to user or Grid scheduler for making scheduling decisions and to execute application in it.

4.4 DESIGN AND IMPLEMENTATION

The proposed semantic component is implemented and tested in Grid Computing Laboratory of Anna University Chennai. Necessary software including Globus Toolkit 4.0 and Protégé was successfully installed in all local machines. Various components of Globus were successfully configured and tested for proper operation. Also, the MDS component has been tested properly for aggregating resource information of the local machines. Further, one of the local machines is designated as ‘submission node’ and configured so as to aggregate resource information of all the local machines. The semantic component is installed in this machine which includes semantic description and discovery module. Ontology template is also maintained in
this machine. The process of creating ontology template and implementation aspect of discovery mechanism is explained in detail in the following subsections.

### 4.4.1 Creation of Knowledge base

Ontology template has been created by considering different computing resources in the Grid. The concept of these resources has been defined properly using relations and properties so that the characteristics of any resource can be defined by its properties. The resource description module accesses the Grid nodes and retrieves resource information by executing ldap queries on those nodes and updates into the ontology template. For every type of information retrieved from the Grid node, the module creates instance of appropriate concept in the ontology template forming conformity relation between instances and their respective concept types. Further, an instance will be exactly related to only one type of concept. Also, the values of various properties retrieved from MDS are assigned to respective properties of the appropriate concepts in the ontology template.

At this point, the ontology template with concepts and properties and corresponding instances and property values together constitutes knowledge base of the Grid resources. To create such knowledge base, Protégé-OWL libraries are used. The libraries contain necessary APIs to create instance for a concept, assign values to properties and other ontology operations. This semantic description of resources facilitates the use of inference engine to interact with the knowledge base and retrieves information semantically. Moreover, the description module is made to execute periodically so that addition and removal of resources is updated in the knowledge base dynamically. A portion of knowledge base is shown in Figure 4.4.
4.4.2 Resource Discovery Mechanism

The discovery module relies on the power of Algernon inference engine. A query tag with the format label:label_value is considered in which the properties of the resource are denoted as label and requested value as label_value. It may also include operators in query label for flexible querying.

For instance, if the user wants to search for machines with free RAM greater than 200 MB, the query should be RAM:>200. Currently, the system supports >, <, = and also NOT operators. Also, the query mechanism allows querying a resource with multiple constraints. For example, if the user
wants to query a machine with free RAM of 200MB and free hard disk space of 10000MB, then the query “freeRAM:200 freeHDD:10000” will retrieve all resources with 200MB and hard disk space with 10000MB. The Query generator module parses the user query using regular expression, stores left tag and right tag in a vector and converts it into suitable Algernon query. For example, the query “freeRAM:200” will be converted into the following machine understandable query.

((instance RAM ?inst)(hasFreeMB ?inst ?val) (:TEST(:LISP(=?val"+rightTag+"))) (presentInComputer?inst ?instance Computer)). Similarly, complex queries can also be formed as follows:


"((:instance RAM ?inst)(hasFreeMB ?inst ?val)(:TEST (:LISP "+rightTag.charAt(0)" ?val "+rightTag.substring(1)+")))(presentInComputer ?inst ?instanceComputer))";


"((:instance CPU ?inst)(hasL2Cache ?inst ?val)(:TEST (:LISP ("+rightTag.charAt(0)+" ?val "+rightTag.substring(1)+" ) ))(:instance WorkStation ?instanceComputer)(hasCPU ?instanceComputer ?inst))";

The discovery module executes the query over the knowledge base of the Grid and obtains the resource that is matching with the user’s request. It not only discovers the resource that exactly matches with that of the request, it
also retrieves resources that exhibit subsumption relation when the exact match is not found. For example, if the user requests a computer with IRIX Operating system, and if the knowledge base does not possess instances of IRIX based machine, the discovery module retrieves all Unix based resources. This is because, the Algernon reasoner infers from the ontology template that IRIX is a subconcept of Unix that is, “≤ (Unix, IRIX) = true” as shown in Figure 4.4. Hence, the discovery module retrieves instances of Unix as it is compatible with IRIX Operating System.

4.5 TESTING WITH GRIDBUS BROKER

The Gridbus broker has been developed by the University of Melbourne as part of the Gridbus Project. It follows a service-oriented architecture and is designed on object-oriented principles with a focus on the idea of promoting simplicity, modularity, reusability, extensibility and flexibility. The broker has been designed to operate with different Grid middleware framework and toolkits such as Globus that primarily runs on Unix-class machines and Alchemi, which is a .NET based Grid computing platform for Microsoft Windows enabled computers. Recently it has been extended to work with Cloud Computing technologies such as Aneka and Amazon EC2. Hence it is possible to create a cross-platform Grid implementation using the Gridbus broker. However, it does not possess its own resource monitoring and discovery mechanism and relies on external information services such as GIIS or GMD as a resource repository. Hence, the semantic component is integrated with the broker in discovering suitable resources based on the semantics of the request for executing user’s jobs.
**Job Descriptor**

The resource discovery module interacts with the knowledge base and obtains best possible resource depending on the user requirements. The user is then prompted to submit the job to the Gridbus broker. Gridbus broker executes user’s job in the resource discovered by the discovery module.

![Diagram of Job Descriptor integration with Gridbus Broker](image)

**Figure 4.5 Integration with Gridbus Broker**

The user is prompted to load the executable followed by the command to execute the job. With this information, a component called the job descriptor creates two files, namely the Application Description File and Resource Description File which are needed by the broker to locate the resource and execute the submitted application as shown in Figure 4.5.

The Application Description File is an XML file with special elements as defined in the XML schema that comes with the broker. XPML supports description of parameter sweep application execution model in which the same application is run for different values of input parameters.
often expressed as ranges. An application description consists of three sections: Parameters, Tasks, and Requirements.

- Parameters normally have a name, type and domain and any additional attributes. Parameters can be of various types including: integer, string, Gridfile and belong to a “domain” such as single, range or file.

- A task consists of “commands” such as copy, execute, substitute etc. The copy command specifies a copy operation to be performed. Each of the copy commands has a source and destination file specified. An execute command is where actual execution happens. The execute command specifies an executable to be run on the remote node. It also specifies any arguments to be passed to the command on the command-line. A substitute command specifies a string substitution inside a text file. This operation is used to substitute the names of user-defined variables.

The resource description file is just an XML file describing the resources that can be used by the broker, and their properties as defined in the resource description schema that comes with the broker. The resource description can be used to describe two types of entities – resources and credentials to access the resources. A resource can be of three types: Compute resources, storage resource and services.

Compute resources are servers to which the user’s jobs can be submitted for execution. Storage resources are used to store the results of execution, and hence can be considered as data sinks. Service resources are those which provide generic services that can be used by the broker. A compute resource is associated with a “domain” which can take two values –
“local” and “remote”. Local resources could be the local computer, or a cluster (on which the broker is running). Remote compute resources are used to represent nodes on the Grid which have a job-submission interface accessible via a network. So resources which run Grid-middleware such as Globus, Unicore and Alchemi are described here.

A storage resource is a data sink where the user can opt to store the results of execution of a Grid application. Currently, this feature is not fully supported by the broker. A service resource can be of two types – “information” services and “application” services. Information services are typically entities which provide information about other resources or services. Application services provide applications hosted on nodes that can be accessed as a service. A “credentials” entry describes the user’s authentication information that is used to access the services provided by a Grid resource. Credential can be of the following types – X.509 based proxy certificates, simple username/password pairs, MyProxy saved proxies or key stores.

For demonstration purpose, we wrote a simple application that performs multiplication of two numbers without taking any external arguments. This job must be executed in a resource identified by the semantic discovery module. The application has been compiled successfully and respective class file is created. The user searches for the resources through the semantic component and the resource discovery module discovers the suitable resource, providing the job descriptor with the hostname of the resource. Meanwhile, the user’s job and the Unix command needed to execute the job are submitted to descriptor. Hence, the class file, command to execute the class file and the hostname of the resource are inputs to the job descriptor. The Job descriptor is implemented using java programming language that automatically creates application description file and resource description file with its input information. They will be submitted to the broker and initiates
scheduling of jobs. Once the execution is over, the results will be collected and presented to the user.

The following is the resource description file if the node discovered by the resource discovery module is the same in which broker is running. In that case, the compute resource is local resource.

```xml
<?xml version="1.0" encoding="UTF-8"?><xgrl
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="../xml/ResourceDescriptionSchema.xsd"
 >

The following snippet identifies credential needed to access the local resource.

<credentials id="prox1" type="proxyCertificate">
 <proxyCertificate source="local">
  <local password=""/>
 </proxyCertificate>
</credentials>

The following snippet identifies the local computing resources. In this case “g06.Grid” is the hostname of the local compute resource.

<resource type="compute" credential="prox1">
 <compute domain="local">
  <local middleware="globus" version =4.0>
   <globus hostname="g06.Grid"/>
  </local>
 </compute>
</resource>
</xgrl>
```
The following is the snippet of the resource description file needed to execute broker in the remote node.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xgrl
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="../xml/ResourceDescriptionSchema.xsd"
 >

 The following snippet identifies credential needed to access the local resource. Here the password has been specified which will be used for authentication purposes.

 <credentials id="prox1" type="proxyCertificate">
  <proxyCertificate source="local">
   <local password="globus"/>
  </proxyCertificate>
 </credentials>

 The following snippet identifies the local computing resources. In this case “g03.Grid” is the hostname of the remote compute resource.

 <resource type="compute" credential="prox1">
  <compute domain="remote">
   <remote middleware="globus" version =4.0>
    <globus hostname="g03.Grid"/>
   </remote>
  </compute>
 </resource>
</xgrl>
```
Similar to the resource description file, the application description will also be created which is purely dependent on the job’s requirement. The structure of the file will change depending on the nature of job and location of the resource.

The following is the portion of the application description file in which the job is to be executed in the local compute resource.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xpml xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation="./xml/XPMLInputSchema.xsd">
  <task type="main">
    <execute location="local">
      <command value="java cals"/>
    </execute>
  </task>
</xpml>
```

Since the ‘execute’ location is local, it is assumed that, the class file available in the directory that the broker can locate. ‘java cals’ is the command to execute the job and the results will be stores in the broker directory.

The following is the portion of the application description file in which the job to be executed in the remote compute resource. In this case, the job file must be transferred from local machine in which broker is running to the remote node. The execution will take place in the remote node and results will be transferred from the remote node to the local machine.
The following snippet is to transfer the class file specified in the source element and copied onto the location of the remote node specified in the destination location.

```xml
<task type="main">
  <copy>
    <source location="local" file="/home/cals.class"/>
    <destination location="node" file="cals.class"/>
  </copy>

  The following snippet is to execute the job which is now present in the remote node.

  <execute location="node">
    <command value="java cals"/>
  </execute>

  Once the execution is over, the results are collected and transferred back to local machine from the remote destination as described by the following snippet.

  <copy>
    <source location="node" file="stdout.$jobname"/>
    <destination location="local" file="/tmp/output"/>
  </copy>
</task>
</xpml>
Once the description files are created, then broker is invoked and scheduling is initialized. The following coding segment invokes the broker with description files as input.

```java
import org.Gridbus.broker.farming.common.GridbusFarmingEngine;
import org.Gridbus.broker.farming.common.BrokerProperties;

public class SimpleBrokerAPIExample {
    public static void main(String[] args) throws Exception {
        try {

            //Create a new "Farming Engine"
            GridbusFarmingEngine fe = new GridbusFarmingEngine(properties);

            //Set the App-description file
            fe.setAppDescriptionFile("/home/adf.xml");

            //Set the Resource-description file
            fe.setResourceDescriptionFile("/home/rdf.xml");

            //Call the initialise method
            fe.init();

            //Start scheduling
            fe.schedule();

            /* The schedule method returns immediately after starting the scheduling. To wait for results / monitor jobs, use the following loop: */
            while (!fe.isSchedulingFinished() && !fe.isSchedulingFailed());

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```
The coding segment waits for results and collects them once the execution is over.

4.6 INTEGRATION WITH GARUDA

To develop Indian eScience infrastructure and to provide Indian researchers, the seamless access of supercomputer-level processing power and knowledge resources, the Department of Information Technology (DIT), Government of India has funded its Centre for Development of Advanced Computing (CDAC) to deploy nationwide computational Grid called ‘Garuda’. It connects and aggregates high end computational resources from 45 research laboratories spanning over 17 cities across the country (Ram and Ramakrishnan 2006). In such an environment, there exists heterogeneity in the nature of resources and difference in usage policies.

In Garuda, the hardware and software configuration of all the computational resources are stored in a text file. Suitable resources are selected manually from the job submission portal. Further, Garuda can submit jobs to the selected resources in two ways, that is, either to the Moab metascheduler, or, directly to ‘GRAM’ component of the globus middleware installed in the Grid resources. When the number of resources participating in the Grid is more, selecting them manually becomes highly difficult and hence a resource discovery mechanism was highly demanded.

Garuda required only the components to construct ontology representation of Grid resources and an interface to discover the resources that matches the user’s requirements. First of all, the ontology of Garuda Grid resources was created using protégé editor. It establishes semantic relationship between all types of computational resources in the Garuda Grid. One of the main requirements of Garuda applications is that an application compiled for lower release of the operating system can also be executed in
higher releases. With this notion, the ontology template is constructed so that
the higher releases were modelled as subconcept of lower releases.

The resource information present in the text file contains the operating system, release, location, hostname, and processor type. The semantic description module parses the resource file and extracts the resource information. These details are read sequentially and modelled as instances and appropriate slots in the ontology template. This creates knowledge base that supports semantic retrieval of information.

Figure 4.6 Semantic Component in Garuda

The discovery module provides an user interface to submit resource request in terms of operating system, and two other variables referring the release and the search level. The search level can be one of two values 0 and 1 that refers to the kind of search the user wants to perform while discovering
suitable resource. The search level 0 refers to the direct match, that is, the resource that exactly matches with the user’s requests are to be discovered. The search level 1 retrieves resources that match closely with the requested, called Compatible match. In this case, the discovery module retrieves the resources information that matches exactly with the request as well as the ones that are closely matching with the request, that is, those exhibits subsumption relation with the request. The GUI used for submitting resource request is shown in Figure 4.6.

In this chapter, an ontology based approach for representation of Grid resources and their discovery based on the semantics present in the resource request has been proposed. It considers hardware information of computational resource for matchmaking. This approach has been extended in the next chapter for considering Grid resource usage policies and their representation using ontology. An ontology based matchmaking system is proposed to determine the Grid resources whose usage policy is matching against that of the requested.