PROOF

B15D

©Saxe-Coburg Publications, 2011.
Trends in Parallel, Distributed, Grid and
Cloud Computing for Engineering
P. Iványi and B.H.V. Topping, (Editors)
Saxe-Coburg Publications, Stirlingshire, Scotland, ~~888-888~~.

Chapter ~~?~~

DRAFT DRAFT DRAFT

Ontology for Contract Negotiations in an Agent-based Grid Resource Management System

M. Drozdowicz[1], K. Wasilewska[1], M. Ganzha[1,2], M. Paprzycki[1,3], N. Attaoui[4], I. Lirkov[5], R. Olejnik[6], D. Petcu[7] and C. Badica[8]
[1]Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
[2]University of Gdansk, Poland, [3]Warsaw Management Academy, Poland
[4]University of Tetouan, Morocco
[5]Bulgarian Academy of Sciences, Sofia, Bulgaria, [6]CNRS, Lille, France
[7]Western University of Timisoara, Romania, ~~7~~University of Craiova, Romania 8

Abstract

It is often claimed that software agents can become intelligent middleware facilitating high-level economy-based resource management in the grid [1]. Furthermore, an argument can be put forward that all meta-level information in such system could be ontology-based, and semantically processed [2]. Most importantly, this information can be used in all forms of contract negotiations, which are needed to introduce grids to business environments. To facilitate ontologically-driven resource management in the grid, an existing grid ontology (CoreGRID) has been modified and extended. The aim of the chapter is to summarize the reasoning that led us to the development of our ontology, and to present and discuss its main features.

Keywords: software agents, grid, resource brokering and management, ontology, semantic information processing.

PROOF

1 Introduction

This chapter is concerned with the development of agent-based grid middleware, in which (a) agents will work in teams (each team will be managed by the *LMaster* agent, (b) all meta-information will be ontologically demarcated and semantically processed (with all team information stored in and managed by the *Client Information Center* (*CIC*) infrastructure being represented by the *CIC* agent), and (c) an economic model will be based on autonomic *Service Level Agreement* (*SLA*) negotiations and *Quality of Service* (*QoS*) monitoring. More information about various aspects of the proposed solution can be found in [3–8], which should be consulted for omitted details. The aim of this chapter is to present the proposed grid ontology, which was developed for the purpose of our system. To achieve this goal, let us start with describing in some detail two major user case scenarios that became the driving force behind this ontology development: (1) "selling" a resource to the team for use in computation, (2) "buying" resources for job execution. Note that in our work we are primarily concerned with computational grids (as the main application area for grid computing of today) rather than, for instance, data grids. However, the proposed ontology should also be applicable to the other types of grid almost without modifications.

The first scenario involves a *User* wishing to make their resource available for use by a team. Extending this scenario, to cover also the process of the team advertising its need for additional worker(s), we can devise the following sequence of actions (see, also [9], Figure 2):

1. The *LMaster* sends a message to the *Client Information Center* (*CIC*), stating that the team is interested in accepting additional worker(s), and specifying conditions (resources) that the worker should bring to the team.
2. *User* interacts with their *LAgent*, specifying the resource they would like to sell to a team.
3. The *LAgent* communicates with the *CIC*, sending a description of offered resources and receiving a list of teams that may be interested in it.
4. The *LAgent* sends a call for proposal (CFP) to the *LMaster*s of teams acquired from the *CIC* describing resources it represents and proposed contract details.
5. The *LAgent* and *LMasters* that respond to the CFP negotiate conditions of the worker contract (for more details about negotiations, see [9]).
6. Assuming that the negotiations resulted in a successful establishment of the *Service Level Agreement* with one of the *LMaster*s, the *LAgent* assumes the role of a worker in the team.

The second scenario describes a case of a *User* who wishes to have a job executed using resources managed by one of the teams. It can be outlined as follows (see, also [9], Figure 3):

1. *User* interacts with their *LAgent*, specifying a set of requirements a resource should fulfil to be able to execute the job.
2. The *LAgent* communicates with the *CIC*, sending the resource constraints and receiving a list of teams owning the resources satisfying its conditions.
3. The *LAgent* sends a call for proposal (CFP) to the selected *LMaster*s and negotiates the job execution contract conditions—an SLA (again, for more details about negotiations, see [9]).
4. If an SLA has been agreed, the *LAgent* sends the job details and necessary data to the contracted team.

For these two scenarios, ontologies have to provide the necessary vocabulary to describe all concepts required, both on the level of inter-agent communication, and for storing information in agents' knowledge bases. To start with, we can notice a significant overlap in the set of ontology concepts used in both scenarios. They both rely heavily on concepts describing resources, or resource requirements, which will be named an *AiG Grid Ontology*. We will expand on this part of the ontology in Section 2.1. Moreover, in both cases there is a strong requirement for concepts used in contract negotiations and related to contract conditions between the participating parties (agents). This is what will constitute the *AiG Conditions Ontology* (described in section 2.3. The last remaining piece of the *AiG Ontology* is a set of classes representing messages exchanged by the agents—the foundation of the communication and negotiation protocols in the system (described in Section 2.4). Here, let us stress that the proposed ontology has been developed with the agent-based resource brokering system in mind. Therefore, it extends concepts that are purely grid-based (that are only a slight modification of the CoreGRID ontology) by a set of concepts and classes that make autonomous negotiations leading to an SLA formation possible.

When considering the two user case scenarios, it is important to notice the key difference between the ways that ontology is used in them. The same concepts can be used either to describe specific entities in the system, or as a means to specify requirements or constraints on these entities. As an example let us consider use of concepts representing a computing resource. In the "selling a resource" scenario, the agent representing the resource owner uses the *AiG Grid Ontology* to describe the particular resource they offers (*i.e.* "their computer"). On the other hand, in the "buying a resource" scenario the *User* specifies the constraints that the team's resources need to satisfy in order to be able to complete the task (*i.e.* the minimal hardware configuration that will be capable of completing the job). Our solution to issues involved in using ontological terms for describing such constraints is described in Section 2.2. In this context, let us note that while the proposed solution to the constraints problem uses properties introduced in OWL 2.0, the remaining parts of the proposed grid resource brokering ontology do not require them.

2 Agents in grid ontology

2.1 Describing grid resources

Let us start by describing in some details the developed *AiG Grid Ontology* (the complete ontology can be found at [10]). Its goal is to provide a set of concepts that describe the infrastructure of the grid and enable detailed specification of grid resources. As seen in the above scenarios, we need a vocabulary for a very detailed technical specification of grid resources: (i) available through the members of the team to execute jobs, and (ii) offered to the team by a potential worker.

Our earlier research showed that a number of grid resource description formats, including ontologies, already exist, and gave us hope for ontology reuse. As a result of an extensive analysis (see, [11], for more details), we realized that the ontology closest to our requirements is the *Core Grid Ontology* (developed as a part of the CoreGRID project [12], and denoted by CGO in what follows). Unfortunately, we found it somewhat lacking in describing the hardware and software configuration of grid resources. Therefore, we decided to extend it with additional classes and properties required by our user case scenarios. Furthermore, our work was focused on improving the ease of interacting with the ontology from the Java code (agents used in our system are Java-based), which resulted in further modifications.

The description of the CoreGrid ontology can be found in [13], therefore in this section we will focus on the classes and properties (both from the *CGO* and the *AiG Grid Ontology*) that are the most important in our user cases, highlighting the changes, and reasons why they were needed. The *CGO* contains a multitude of concepts related to the structure, organization and configuration of grid elements, but for the task of managing resources and scheduling jobs the most important are these contained in the hierarchies of the *GridComponent* and the *GridResource* classes (see Figure 1). Following is a detailed description of this part of the ontology. Newly added concepts are highlighted in bold font.

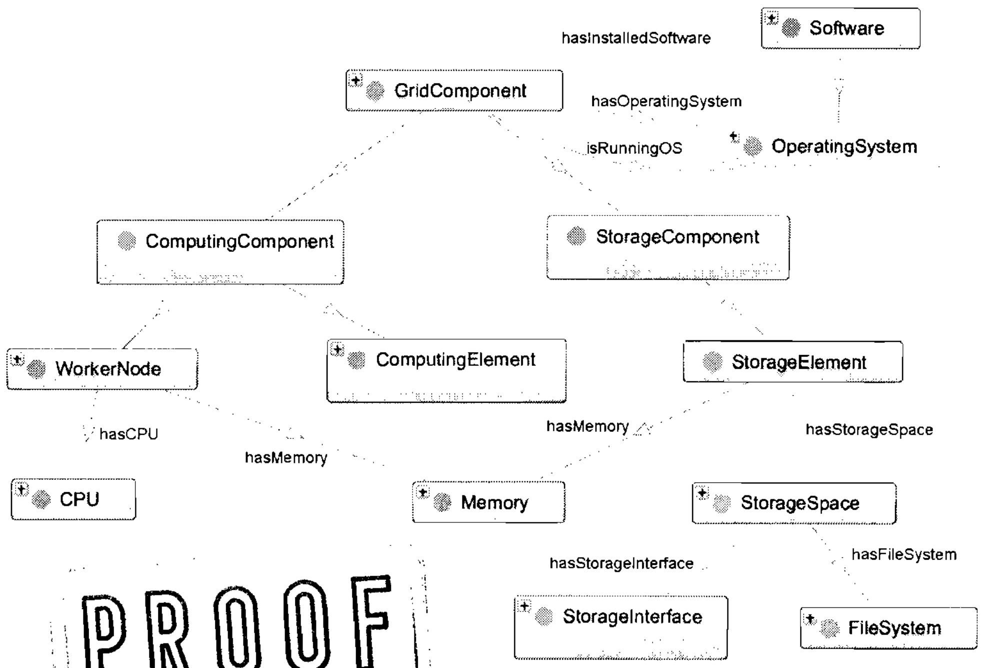

Figure 1: Resource concepts in grid ontology

- *GridComponent:* the base of all classes describing physical or virtual elements of the grid, providing access to the grid resources, and hosting grid services.

- *ComputingComponent:* a subclass of the *GridComponent* that provides access to computing resources. It is the domain of properties such as:
  - ***hasInstalledSoftware*:** defines the software configuration of any grid component using the *Software* class.
  - ***hasOperatingSystem*:** specifies an operating system installed and available on the grid component using the *OperatingSystem* class.
  - ***isRunningOS*:** the *OperatingSystem* currently running on the grid component.
- *WorkerNode:* a *ComputingComponent* that is able to provide access to its computing power. It has the following properties:
  - ***hasCPU:*** An instance of the *CPU* class, describing the processing unit exposed by the component.
  - ***hasMemory*:** The memory available for use by the services accessing the *WorkerNode*.
  - ***isVirtualized*:** Property stating if the component is a physical or a virtual machine.
- *StorageElement:* any *GridComponent* that exposes storage for use by services and grid users. Linked to the *StorageSpace* class using the ***hasStorageSpace*** property. *GridResource* the other important base class of the *CGO* ontology, *GridResource* represents all computing, storage, networking, hardware and software resources that can be used by services running on the grid.
- *ComputingElement:* A *ComputingComponent* that manages a set of *WorkerNodes*. It can be described using the following properties:
  - ***hasWN:*** links to a *WorkerNode* managed by the *ComputingElement*.
  - ***hasQueue:*** *Queue* used by the component for storing jobs to be executed.
  - ***runningServices:*** services running on the component to handle tasks such as job management *etc.*

- *CPU:* subclass of *ComputingResource*, the processing unit of a grid component, described using the following properties:
  - ***hasCores*:** number of processor cores.
  - ***hasL1CacheSize*, *hasL2CacheSize*, *hasL3CacheSize*:** size of the internal processor caches.
  - ***hasClockSpeed*:** the clock speed of the processor. Property replaces the *clockSpeed* property from the *CGO*, because *clockSpeed* was defined with a range of *string*, while for reasoning, in our user cases, we need a numeric data type.

PROOF

- ◦ ***hasArchitecture*:** the architecture of the system expressed using the ***CPU-Architecture*** class.
  - ◦ ***hasVendor*:** the producer of the CPU as a link to the ***CPUVendor*** class.
  - ◦ ***hasModelName*:** the exact model name of the CPU. Property replaces the *modelName* property from the *CGO* to provide a domain specification to its definition which provides us with better explorability of the ontology, especially for processing by agents.

- *Memory:* the class representing the memory of a grid component. In the *AiG Grid Ontology* two subclasses of this type have been introduced: ***PhysicalMemory*** and ***VirtualMemory***. *Memory* can be described using the following properties:
  - ◦ ***hasAvailableSize*:** the amount in megabytes of the current free memory.
  - ◦ ***hasTotalSize*:** the total amount of memory installed on the machine.

- *Software:* class representing a piece of software that can be installed on a machine. The existing subclasses include: *OperatingSystem*, *Exec*, *Lib* but can be extended when the need arises. Each instance of *Software* can be described using the following properties:
  - ◦ *hasName:* the product name of the software.
  - ◦ ***hasVersion*:** the official version number of the software.

- *StorageResource:* a subclass of *GridResource* and a superclass of concepts related to access to the storage available on grid components.

- *FileSystem:* a subclass of *StorageResource*, representing the type of file system in which the storage is formatted. Currently available subclasses are: *LINUX-EXT3* and *WINNT*.

- *StorageInterface:* the interface for connecting mass storage devices to computers. The subclasses currently included in the ontology are *IDE*, *SCSI* and ***SATA***.

- *StorageSpace:* the main class used for representing the storage available in the grid. It uses the other aforementioned *StorageResource* subclasses through the ***hasFileSystem*** and the ***hasStorageInterface*** properties. Other properties that describe it include:
  - ◦ ***hasAvailableSize*:** the current available storage space in megabytes.
  - ◦ ***hasTotalSize*:** the total capacity of the storage resource in megabytes.

PROOF

Another minor change we needed to make to the *CGO* ontology resulted from some consistency issues when attempting to classify the ontology using the Protege 4.1 default reasoners (HermiT and Fact++). We found that some individuals defined in the ontology had datatype property values inconsistent with the range definition of the properties. These included the properties *hasName*, *hasModel*, *clockspeed*, the range of which was “&xsd;string” but in the case of some individuals the value was set as an XML literal—“xml:lang=en”. The issue with classifying the ontology was probably caused by the introduction of new data types in OWL 2.0, which is the OWL version used by the reasoners. Therefore, we changed the value type of these individuals' properties to “&xsd;string” to fix the issue.

2.1.1 Example 1

Let us now look at two examples of resource descriptions using our *AiG Grid Ontology*. First, a single PC is available to the grid users. It is running the Windows Vista operating system and is equipped with an Intel Core 2 Duo processor, 4 GB of memory of which 2.5 GB is available to the users, and 300 GB of storage space on an NTFS partition.

The desktop computer itself is of types *WorkerNode* and *StorageElement*, because it exposes computing capabilities as well as the storage space. Using the ontology it would be described as shown in the following listing.

```
:desktopWorkerNode1
   a cgo:WorkerNode , cgo:StorageElement ;
   :hasMemory [
      a :PhysicalMemory ;
      :hasAvailableSize "2500"^^xsd:int ;
      :hasTotalSize "4000"^^xsd:int .
   ];
   :hasStorageSpace [
      a cgo:StorageSpace ;
      :hasAvailableSize "25000"^^xsd:int ;
      :hasFileSystem :winnt ;
      :hasStorageInterface
            :ide ;
      :hasTotalSize "300000"^^xsd:int .
   ];
   :isRunningOS :vista_sp2 ;
   :isVirtualized "false"^^xsd:boolean ;
   cgo:hasCPU :intel_core2duo_e8300 .

:vista_sp2
   a cgo:Windows ;
   :hasVersion "6.0.6002.18005.090410-1830"^^xsd:string ;
   cgo:hasName "Windows Vista Service Pack 2"^^xsd:string .

:intel_core2duo_e8300
   a cgo:CPU ;
   :hasArchitecture :Intel64 ;
   :hasClockSpeed "2830"^^xsd:int ;
   :hasCores "2"^^xsd:int ;
   :hasL2CacheSize "6000"^^xsd:int ;
   :hasModelName "Intel Core 2 Duo Processor E8300"^^xsd:string ;
   :hasVendor :Intel ;
   cgo:hasName "Intel  Core 2 Duo Processor E8300
      (6M Cache, 2.83 GHz, 1333 MHz FSB)"^^xsd:string .
```

PROOF

2.1.2 Example 2

Let us now consider a more complex scenario. We would like to represent a computing grid element consisting of two virtualized machines working together under the control of the Condor middleware [14]. They both use the Debian Linux operating system and have a similar hardware configuration. The representation of such a computing system is an instance of the *ComputingElement* class.

The *WorkerNode* instances representing the components of such an "aggregate grid node" would be described as in the following listing.

```
:compositeWorker
   a cgo:ComputingElement ;
   :isRunningOS :debian_5.0 ;
   cgo:hasWN :condorWorkerNode1 , :condorWorkerNode2 ;
   cgo:runningServices
      :condor .

:condorWorkerNode1
   a cgo:WorkerNode ;
   :hasMemory [
      a :PhysicalMemory ;
      :hasAvailableSize "1500"^^xsd:int ;
      :hasTotalSize "3000"^^xsd:int
   ];
   :hasStorageSpace [
      a cgo:StorageSpace ;
      :hasAvailableSize "120000"^^xsd:int ;
      :hasFileSystem :ext3 ;
      :hasStorageInterface
         :ide ;
      :hasTotalSize "350000"^^xsd:int
   ];
   :isRunningOS :debian_5.0 ;
   :isVirtualized "true"^^xsd:boolean ;
   cgo:hasCPU :intel_xeon_e7430 .

:condorWorkerNode2
   a cgo:WorkerNode ;
   :hasMemory [
      a :PhysicalMemory ;
      :hasAvailableSize "2000"^^xsd:int ;
      :hasTotalSize "3000"^^xsd:int
   ];
   :hasStorageSpace [
      a owl:Thing , owl:NamedIndividual ;
      :hasAvailableSize "70000"^^xsd:int ;
      :hasFileSystem :ext3 ;
      :hasStorageInterface
         :ide ;
      :hasTotalSize "250000"^^xsd:int
   ];
   :isRunningOS :debian_5.0 ;
   :isVirtualized "true"^^xsd:boolean ;
   cgo:hasCPU :intel_xeon_e7430 .

:debian_5.0
   a cgo:Linux ;
   :hasName "Debian GNU/Linux 5.0 ( lenny )"^^xsd:string ;
   :hasVersion "5.0"^^xsd:string .

:intel_xeon_e7430
   a cgo:CPU ;
```

PROOF

```
:hasArchitecture :Intel64 ;
:hasClockSpeed "2130"^^xsd:int ;
:hasCores "4"^^xsd:int ;
:hasL2CacheSize "12000"^^xsd:int ;
:hasModelName "Intel Xeon Processor E7430"^^xsd:string ;
:hasName "Intel Xeon Processor E7430
   (12M Cache, 2.13 GHz, 1066 MHz FSB)"^^xsd:string ;
:hasVendor :Intel .
```

PROOF

2.2 Modeling requirements and constraints

Let us now consider the problem that is essential for effective use of our ontology in the *AiG* system: matching resources to resource requirements. As follows from the two user case scenarios, there are two situations when this is necessary:

1. When a *User* wishes to sell a resource to a team, its *LAgent* sends to the *CIC* a specific description of the resource it offers. The *CIC* matches this description against a collection of resource requirements received from the registered teams and returns a list of teams seeking such a worker.

2. When a *User* wishes to complete a job, its *LAgent* sends to the *CIC* a message containing constraints on resource description, limiting the resource configuration space to these that are capable of executing the job. In this case, the *CIC* needs to find a list of resources satisfying the received requirements.

We can see that these two cases are very similar, they deal with matching resource description(s) and a set of resource constraints (requirements), but the matching itself proceeds from "opposite directions." In the first case, a description of a resource is matched against a set of requirements (constraints), while in the second case, a definition of requirements (constraints) is matched against a set of descriptions. In both cases we need to describe the constraints imposed on a resource in a way that enables matching with resource description. Moreover, to support the first scenario, giving the *CIC* the ability of gathering the team requirements, it is necessary that this constraint representation is easy to store in a knowledge base.

As an example, let us consider a case where a *User* expresses a demand that the job should be executed on a machine that is running the Debian Linux operating system, has a quad-core CPU of at least 2.5 GHz and a minimum of 4 GB of physical memory. As one can easily understand, this is a different task from merely specifying the values of the properties. Here, we deal with value ranges (minimum, maximum, between) as well as logical operators between different conditions (a machine with faster CPU **or** with more memory).

Our solution takes advantage of the class expressions and datatype restrictions introduced in OWL 2.0. Specifically, individuals satisfying some constraints can be expressed as an OWL class defined using restrictions on the properties that describe it. To illustrate it with a simple example let us consider a case where we would like to define a set of CPUs that have a clock speed greater than 2.5 GHz. We would express it using the following OWL class:

PROOF

```
:RequiredCPU
   a owl:Class ;
   owl:intersectionOf (
      cgo:CPU
      [
         a owl:Restriction ;
         owl:onProperty aiggo:hasClockSpeed ;
         owl:someValuesFrom [
            a rdfs:Datatype ;
            owl:onDatatype xsd:int ;
            owl:withRestrictions
               ([ xsd:minInclusive 2500])
         ]
      ]
   ) .
```

In the snippet we can also see an example of using the datatype restriction introduced in OWL 2.0 in the form of the *minInclusive* element. Such a constraint class can then be used in a definition of a class representing *WorkerNode*s that are necessary to perform a job, as in the following listing.

```
:RequiredResource
   a owl:Class ;
   rdfs:subClassOf cgo:WorkerNode ;
   owl:equivalentClass [
      a owl:Restriction ;
      owl:onProperty cgo:hasCPU ;
      owl:someValuesFrom :RequiredCPU
   ] .
```

Using this approach, the task of finding a list of individuals satisfying the requirements simplifies to using an OWL 2.0 capable reasoner (*e.g.* Fact++ [15] or HermiT [16]), to classify the knowledge base and infer which individuals are of the *RequiredResource* class. Since the class expressions system in OWL enable mixing restrictions on multiple properties, by using this approach one will be able to construct constrains even for complex use cases.

Another challenge was to assign weights to the defined constraints so that we can not only restrict the individuals but also rank them according to our requirements. The proposed solution is to take advantage of the OWL annotations system as a means of linking the weight to the constraint. Specifically, we use a special annotation: *constraintWeight*, defined in the *ContractConditions* that we will assign to the aforementioned constraint class descriptions. Naturally, the reasoner will not be able to make use of this annotation when processing the knowledge base. Therefore, a custom component will post-process the result of the inference to rank matched individuals. In our implementation we will interpret the lack of the *constraintWeight* annotation on a constraint class as a weight of one for that property.

Let us illustrate this process with an example. We will extend the requirements placed on a *GridResource* by introducing an additional constraint on the available physical memory. Moreover, we would like to make sure that this amount is valued more (*i.e.* is more important to the user) than the clock speed of the CPU. We would express this set of requirements as follows.

```
:RequiredMemory
   a owl:Class ;
   aigco:constraintWeight "5"^^xsd:float ;
```

```
owl:equivalentClass [
    a owl:Class ;
    owl:intersectionOf (
        cgo:Memory
        [
            a owl:Restriction ;
            owl:onProperty aiggo:hasAvailableSize ;
            owl:someValuesFrom [
                a rdfs:Datatype ;
                owl:onDatatype xsd:int ;
                owl:withRestrictions
                    ([ xsd:minInclusive 3000 ])
            ]
        ]
    )
] .
```

PROOF

To address this requirement, the *RequiredResource* class was modified in the following way.

```
:RequiredResource
    a owl:Class ;
    rdfs:subClassOf cgo:WorkerNode ;
    owl:equivalentClass [
        owl:intersectionOf (
            [
                a owl:Restriction ;
                owl:onProperty cgo:hasCPU ;
                owl:someValuesFrom :RequiredCPU
            ]
            [
                a owl:Restriction ;
                owl:onProperty aiggo:hasMemory ;
                owl:someValuesFrom :RequiredMemory
            ]
        )
    ] .
```

PROOF

As mentioned before, we accept constraint definitions with no weight annotation, therefore, no further changes are needed to the *RequiredCPU* class.

2.3 Describing contract conditions

The conditions of contracts between the agents, both in the case of a resource joining the team, and a user contracting the team to execute a job on its resources, are described using the *AiG Conditions Ontology*. It consists of two parts. First, it uses the vocabulary from the *AiG Grid Ontology* to describe owned, or used, grid resources. Second, it contains an added terminology, covering availability, quality of service and payment terms. These terms will be needed in the SLA negotiations (see, also [9]).

Here, the root classes are: *WorkerContractConditions* and *JobExecutionConditions*; used, respectively, during "selling resource" and "buying resource" scenarios.

The ontology supports multiple pricing options, using the *Pricing* class, several payment methods, through the *PaymentMechanism* class, and charging the user, based on the real usage of various parameters, thanks to the hierarchy of the *ServiceChargableItem*. Let us start by describing these core concepts in more detail.

- *PaymentConditions:* the class that defines complete information about the contract conditions concerning payment for the services either of the team, or of the worker. The following properties can be used to describe it:
  - *offeredChargableItem:* a set of *ServiceChargableItem* instances that describe the various commodities (e.g. memory, storage space, CPU time) that can be utilized during job execution, along with their prices per unit. The commodities mentioned here are represented using the classes from the *AiG Grid Ontology.*
  - *fixedAvailabilityPrice* and *fixedUtilizationPrice:* additional properties describing prices per hour of resource availability and utilization, independent on the way the resource is used. They can be used to define the job execution cost combined with the chargeable items, or instead of them—if the cost calculation model is simple.
  - *leadTime:* the time, in hours, during which the payment should be made after completing the job execution.
  - *delayPenalty:* the amount of money that needs to be paid if the payment deadline specified by *leadTime* is not observed.
  - *paymentMechanism:* the mechanism of payment accepted by the sides of the agreement. The possible values of this property are defined as subclasses of the *PaymentMechanism* class described below.

- *ServiceChargableItem:* the description of a commodity offered as a part of the contract, for utilization during job execution, including its pricing information and the grid component that was used. The information that the class can be described with, include:
  - *availabilityPrice:* the price that will have to be paid for the availability of the commodity. The domain of this property is not a simple datatype, but the *Pricing* class that enables a much more robust definition of the pricing conditions.
  - *utilizationPrice:* the price of a unit of utilization of a commodity.
  - *usedResource:* the *GridResource* which the service utilized, or that was reserved for availability.
  - *usedComponent:* the *GridComponent* which the service utilized, or that was reserved for availability.

  The importance of the *ServiceChargableItems* concept is in its subclass hierarchy, which is built of the following items:
  - *MemoryUsageChargableItem:* the memory used when serving the user.
  - *NetworkBandwithChargableItem:* the network bandwidth used by the service; further split into: *DownloadChargableItem* and *UploadChargableItem* subclasses; for the possibility of an even more fine grained pricing specification.

  - *ProcessingTimeChargableItem:* the processing time spent when serving the user. Can be described also using one of its subclasses: *CPUTimeChargableItem:* pure CPU time used; or *WallClockTimeChargableItem:* the total time including I/O *etc.*
  - *ProcessorNumberChargableItem:* used to specify the pricing of availability and utilization of the number of processors of the used resource.
  - *SoftwareChargableItem:* the price of a software resource required by the user task.
  - *StorageChargableItem:* the price of storage on the Grid resource. This item is split into two sub-items: *TemporaryStorageChargableItem*, which describes the storage that is used only for the time of executing a job and is cleaned afterwards; and *PermanentStorageChargableItem* - storage that should be retained between successive service executions.

- *Pricing:* a class describing the different pricing options depending on the conditions in which the job is executed. The various prices are specified using the following datatype properties:
  - *peakTimePrice:* the price used during peak hours
  - *offPeakTimePrice:* the price used during off–peak hours
  - *holidayTimePrice:* the price effective during holidays
  - *lunchTimePrice:* the price effective during lunch time
  - *discountWhenLightlyLoaded:* enables specifying a discount that applies when the resource's load is less than a certain value (*e.g.* 50%).
  - *raiseWhenHighDemand:* enables specifying an additional price increase that applies when the resource's load is greater than a certain value (*e.g.* 50%).

- *PaymentMechanism:* a hierarchy of classes that describe the accepted mechanisms of paying for the team's, or worker's, services. Currently, it consists of the following subclasses:
  - *GrantsBased:* the payment is not processed directly, but is charged against an account managed by being based on some up front agreement.
  - *PrePaid:* the payment is transferred in advance.
  - *UseAndPayLater:* the payment is made after the job execution has been finished in the time specified by the *leadTime* property of the *PaymentConditions* class.

Let us now turn our attention to the classes used only in either of the two user case scenarios. We will begin by describing the class representing the conditions of a contract between a resource owner and the team.

PROOF

- *WorkerContractConditions:* describes the details of the agreement between a worker and the team. This class is the domain of the following properties:
    - ○ *contractPeriod:* uses the *TemporalEntity* class from the *OWL–Time* ontology to specify the period of time during which the contract is in effect.
    - ○ *contractedResource:* the *GridComponent* (from the *Grid Ontology*) that is the subject of the contract.
    - ○ *guaranteedUtilization:* the percentage of contracted time that the team commits to utilize the contracted resource. In the case of our system, regardless of the real usage of the resource, the utilization of the resource taken into account when calculating the payment will not be less than the value of this property.
    - ○ *isContractExtensionPossible:* specifies whether, after the ending of the *contractPeriod*, the agreement can be extended.
    - ○ *workerAvailability:* the period of time, or a set of such, during which the worker commits to being available for utilization by the team. Defined using the *TemporalEntity* class.
    - ○ *paymentConditions:* the conditions on which the resource owner will be rewarded for its services to the team. The property's range is the *PaymentConditions* class described further on.

We will illustrate the usage of this class in an example of conditions of an extendable agreement between an owner of a grid resource and the team wishing to take advantage of capabilities of that resource—the *desktopWorkerNode* from 2.1. The resource will work for the team five days a week (Monday – Friday) from 18:00 to 8:00 for the next two weeks. The worker will be paid a fixed fee of ten grid tokens for an hour when it is available for use, and forty tokens when it is actively used. Additionally, the user is charged separately for incoming and outgoing bandwidth, used disk space, and CPU time. The owner of the resource receives the payment in a pay-after-use manner at the end of each week.

PROOF

```
:DesktopNodeContract
  a :WorkerContractConditions ;
  :contractPeriod :Dec13Dec24 ;
  :contractedResource aiggo:desktopWorkerNode1 ;
  :isContractExtensionPossible
      "true"^^xsd:boolean ;
  :workerAvailability :Dec13Dec14 , :Dec14Dec15 , :Dec15Dec16 , :Dec16Dec17 ,
      :Dec20Dec21 , :Dec21Dec22 , :Dec22Dec23 , :Dec23Dec24 ;
  :paymentConditions [
    a :PaymentConditions ;
    :fixedAvailabilityPrice [
      a :Pricing ;
      :offPeakTimePrice "10.0"^^xsd:float
    ];
    :fixedUtilizationPrice[
      a :Pricing ;
      :offPeakTimePrice "40.0"^^xsd:float
    ];
    :offeredChargableItem
```

PROOF

```
        [ a :UploadChargableItem ;
           :utilizationPrice [ a :Pricing ; :offPeakTimePrice "4.0"^^xsd:float]
        ] ,
        [ a :DownloadChargableItem ;
           :utilizationPrice [a :Pricing ; :offPeakTimePrice "2.0"^^xsd:float ]
        ],
        [ a :WallClockTimeChargableItem ;
           :utilizationPrice [ a :Pricing ; :offPeakTimePrice "10.0"^^xsd:float ]
        ],
        [ a :TemporaryStorageChargableItem ;
           :utilizationPrice [ a :Pricing ; :offPeakTimePrice "5.0"^^xsd:float ]
        ];
     :paymentMechanism :useAndPayLater .
  ] .

:Dec13Dec24
   a time:ProperInterval ;
   time:hasBeginning [
      a time:Instant ;
      time:inXSDDateTime "2010-12-13T00:00:00"^^xsd:dateTime
   ] ;
   time:hasEnd [
      a time:Instant ;
      time:inXSDDateTime "2010-12-24T23:59:59"^^xsd:dateTime
   ].

:Dec13Dec14
   a time:ProperInterval ;
   time:hasBeginning [
      a time:Instant ;
      time:inXSDDateTime "2010-12-13T18:00:00"^^xsd:dateTime
   ] ;
   time:hasEnd [
      a time:Instant ;
      time:inXSDDateTime "2010-12-14T08:00:00"^^xsd:dateTime
   ].

:Dec14Dec15
   a time:ProperInterval ;
   time:hasBeginning [
      a time:Instant ;
      time:inXSDDateTime "2010-12-14T18:00:00"^^xsd:dateTime
   ] ;
   time:hasEnd [
      a time:Instant ;
      time:inXSDDateTime "2010-12-15T08:00:00"^^xsd:dateTime
   ].

#...
```

PROOF

Let us now move to the second scenario and the *JobExecutionConditions* class. Here we see:

- *JobExecutionConditions:* the conditions of the contract between the *User* scheduling the task for execution and the team accepting the job. The class is described with the following properties:
  - *contractedResource:* the *GridComponent* that will be used to perform the job. Note that this instance does not need to match a physically existing grid resource, but can be used only to describe the agreed hardware or software configuration. Moreover, we expect that in most cases a different solution to specifying the constraints of the requested / contracted grid

resource will be used (see 2.2).

- *jobExecutionTimeline:* the *TemporalEntity* that describes either the period of time during which the job should be executed (including start time and deadline), or simply a time instance specifying the deadline.
- *deadlinePenalty:* the monetary amount that will be paid to the *User* in case of missing the deadline, as specified in the contract.
- *paymentConditions:* the payment related details of the contract. They are specified using the same *PaymentConditions* class as in the case of the *WorkerContractConditions*.

To give an example of its usage let us consider a contract, where the team agrees to finish job execution by the 20th of December 2010 with a penalty for missing the deadline equal to one hundred grid tokens. The team also commits to having the job executed on a composite computing node running the Linux operating system in a Condor environment—the *compositeWorker* from Section 2.1. In return, the other party agrees to pay the team a fee of twenty grid tokens for each hour of CPU wall clock time, as well as 0.002 tokens for each megabyte of physical memory used per hour and 1 for each megabyte transferred over the network, regardless if it is inbound or outbound. Because the computing node is also utilized by the organization owning it, the price for the CPU processing depends on whether it is used during peak or off-peak hours. The lower, off-peak price is eighteen grid tokens per hour. The customer will make a payment within seven days from the job completion. The details of such a contract would be represented as follows.

```
:CondorJobContract
   a :JobExecutionConditions ;
   :contractedResource aiggo:compositeWorker ;
   :deadlinePenalty ”100.0”^^xsd:float ;
   :jobExecutionTimeline [
      a time:Instant ;
      time:inXSDDateTime ”2010-12-20T23:59:59”^^xsd:dateTime
   ];
   :paymentConditions [
      a :PaymentConditions ;
      :leadTime ”2010-12-28T00:00:00”^^xsd:dateTime ;
      :offeredChargableItem
         [ a :MemoryUsageChargableItem ;
            :utilizationPrice [ a :Pricing ;
               :offPeakTimePrice ”3.0”^^xsd:float ;
               :peakTimePrice ”3.0”^^xsd:float
            ]
         ],
         [ a :NetworkBandwithChargableItem ;
            :utilizationPrice [a :Pricing ;
               :offPeakTimePrice ”1.0”^^xsd:float ;
               :peakTimePrice ”1.0”^^xsd:float
            ]
         ],
         [ a :WallClockTimeChargableItem ;
            :utilizationPrice [ a :Pricing ;
               :offPeakTimePrice ”18.0”^^xsd:float ;
               :peakTimePrice ”20.0”^^xsd:float
            ]
         ]
   ].
```

2.4 Ontology of contract negotiation messages

The *AiG Messages Ontology* was created mostly as a basis for the communication between negotiating agents and contains definitions of exchanged messages. Matching the two user case scenarios, the ontology consists of two parallel sets of concepts - related to a worker joining a team (subclasses of *TeamJoingMessage*), and a *User* wanting to execute a job (subclasses of *JobExecutionMessage*). The first part of the ontology comprises the following classes and properties:

- *TeamJoiningMessage:* the superclass of all message classes related to the case of a resource joining a team.
- *TeamEnquiry:* the class describing a message initiating the process of negotiating the terms of joining a team. The hardware and software configuration of the resource wishing to join is described using the *offeredResource* property linking to the *GridComponent* class from the *AiG Grid Ontology*. The message itself may also contain a definition of a class based on the *JobExecutionConditions*, defining the constraints on the contract parameters, such as availability or price (see 2.2).
- *TeamOffer:* the class representing a response to the *TeamEnquiry*. The proposed contract conditions are defined using the *WorkerContractConditions* through the *proposedWorkerContract*.
- *TeamRefusal:* the message sent as a response to the *TeamEnquiry* if the team decides it is not interested in the *LAgent* joining its ranks.
- *TeamOfferAccept:* the class representing a message sent by the *LAgent* stating that it accepts the contract conditions and wants to join the team.
- *TeamOfferReject:* the message sent by the worker stating that it does not accept the contract conditions and wants to break the negotiations.
- *TeamCounterOffer:* the definition of a message sent by the worker in response to *TeamOffer* in which it rejects the offered contract conditions but wishes to continue the negotiations in a multi-round manner, proposing a different set of conditions specified using the *proposedWorkerContract* property. The team should respond to such a proposition using the *TeamOfferAccept*, THE *TeamOfferReject*, or the *TeamCounterOffer* message.
- *JobExecutionMessage:* the superclass of all message classes related to the scenario of a *User* wishing to execute a job in the grid.

The part of the *AiG Messages Ontology* regarding job execution negotiations, consists of the following concepts:

- *JobExecutionEnquiry:* the class representing a message sent by the *User* to the *LMasters* of selected teams, as a call for proposal (CFP) regarding the execution of a job. In our system, along with an instance of this class, a definition of another class will be sent as a means of describing the constraints of the resources necessary to execute the job. This approach is discussed in Section 2.2.

- *JobExecutionOffer:* the response to the job execution CFP, sent by the team as a means of describing the proposed conditions of the contract (instance of *JobExecutionConditions* through the *proposedJobExecutionContract* property) as well as the resources belonging to the team that are available and suitable for the job (instance of *GridComponent* through the *offeredResource* property).

- *JobExecutionRefusal:* the message sent by the team to *User* as a response to the CFP in a case when it is impossible (or not profitable) to execute the job.

- *JobExecutionOfferAccept:* the message informing the team that it has been selected to execute the job on the proposed conditions.

- *JobExecutionOfferReject:* the message informing the team that its offer has been rejected and the *User* is not interested in continuing negotiations.

- *JobExecutionCounterOffer:* the message informing the team that its offer has been rejected, but the *User* would like to propose a different set of conditions. The revised contract is described in the *JobExecutionConditions* class using the *proposedJobExecutionContract*. The team should respond to such proposition using either the *JobExecutionOfferAccept*, the *JobExecutionOfferReject*, or the *JobExecutionCounterOffer*.

3 Concluding remarks

In this chapter we have presented the main features of our ontology designed for a grid resource brokering system, which consists of three main parts, (a) ontology of grid resources, (b) ontology of terms needed in contract negotiations, and (c) messaging ontology. This ontology can be found at [17] and is ready to be tested and applied in development of grid applications. We will use it in our *Agents in Grid* system to facilitate contract negotiations in both the above described user-case scenarios.

Acknowledgement

The work of Maria Ganzha and Michal Drozdowicz was supported by the "Funds for Science" of the Polish Ministry for Science and Higher Education for years 2008-2011, as a research project (contract number N516 382434). The Polish-Bulgarian collaboration was partially supported by the Parallel and Distributed Computing Practices grant. The Polish-French collaboration was partially supported by the PICS grant

New Methods for Balancing Loads and Scheduling Jobs in the Grid and Dedicated Systems. The Polish-Romanian collaboration was partially supported by the Agent-Based Service Negotiation in Computational Grids; and the Agents, Grids and Heterogeneous Computing grants.

References

[1] I. Foster, N.R. Jennings, C. Kesselman, "Brain Meets Brawn: Why Grid and Agents Need Each Other", International Joint Conference on Autonomous Agents and Multiagent Systems, 1, 8–15, 2004.

[2] J. Hendler, "Agents and the Semantic Web", IEEE Intelligent Systems, 16(2), 30–37, 2001.

[3] M. Dominiak, M. Ganzha, M. Gawinecki, W. Kuranowski, M. Paprzycki, S. Margenov, I. Lirkov, "Utilizing Agent Teams in Grid Resource Brokering", International Transactions on Systems Science and Applications, 3(4), 296–306, 2008.

[4] M. Drozdowicz, M. Ganzha, W. Kuranowski, M. Paprzycki, I. Alshabani, R. Olejnik, M. Taifour, M. Senobari, I. Lirkov, "Software Agents in ADAJ: Load Balancing in a Distributed Environment", in M. Todorov, (Editor), "Applications of Mathematics in Engineering and Economics'34", AIP Conf. Proc., 1067,527–540, American Institute of Physics, College Park, MD, 2008.

[5] W. Kuranowski, M. Ganzha, M. Gawinecki, M. Paprzycki, I. Lirkov, S. Margenov, "Forming and managing agent teams acting as resource brokers in the Grid—preliminary considerations", International Journal of Computational Intelligence Research, 4(1), 9–16, 2008.

[6] M. Ganzha, M. Paprzycki, I. Lirkov, "Trust Management in an Agent-based Grid Resource Brokering System—Preliminary Considerations", in M. Todorov, (Editor), "Applications of Mathematics in Engineering and Economics'33", AIP Conf. Proc., 946, 35–46, American Institute of Physics, College Park, MD, 2007.

[7] M. Dominiak, M. Ganzha, M. Paprzycki, "Selecting grid-agent-team to execute user-job—initial solution", in "Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems", 249–256, IEEE CS Press, Los Alamitos, CA, 2007.

[8] W. Kuranowski, M. Paprzycki, M. Ganzha, M. Gawinecki, I. Lirkov, S. Margenov, "Agents as resource brokers in grids—forming agent teams", 4818, 472–480, Springer, Berlin, 2007.

[9] K. Wasielewska, M. Drozdowicz, M. Ganzha, M. Paprzycki, N. Attaui, D. Petcu, C. Badica, R. Olejnik, I. Lirkov, "Negotiations in an Agent-based Grid Resource Brokering Systems", in P. Iványi, B.H.V. Topping, (Editors), "Trends in in Parallel, Distributed, Grid and Cloud Computing for Engineering", Saxe-Coburg Publications, Stirlingshire, UK, 2011.

[10] "Agents in the grid–resource management". `http://sourceforge.net/projects/gridagents`

[11] M. Drozdowicz, M. Ganzha, M. Paprzycki, R. Olejnik, I. Lirkov, P. Telegin, M. Senobari, “Ontologies, Agents and the Grid: An Overview”, in B.H.V. Topping, P. Iványi, (Editors), “Parallel, Distributed and Grid Computing for Engineering”, Saxe-Coburg Publications, Stirlingshire, UK, Chapter 7, 117-140, 2009. doi:10.4203/csets.21.7

[12] W. Xing, M.D. Dikaiakos, R. Sakellariou, S. Orlando, D. Laforenza, “Design and Development of a Core Grid Ontology”, in “Proc. of the CoreGRID Workshop ”Integrated research in Grid Computing”, 21–31, November 2005.

[13] “Core Grid Ontology”. `http://grid.ucy.ac.cy/grisen/cgo.owl`

[14] D. Thain, T. Tannenbaum, M. Livny, “Condor and the Grid”, in F. Berman, G. Fox, T. Hey, (Editors), “Grid Computing: Making the Global Infrastructure a Reality”, John Wiley & Sons Inc., December 2002.

[15] “OWL: FaCT++”. `http://owl.man.ac.uk/factplusplus/`

[16] “HermiT OWL Reasoner”. `http://hermit-reasoner.com/`

[17] “Agents in the Grid Project”. `http://sourceforge.net/projects/gridagents`