Using ontologies to manage resources in Grid computing—practical aspects

Michał Drozdowicz[1], Maria Ganzha[1,2], Katarzyna Wasielewska[1], Marcin Paprzycki[1], and Paweł Szmeja[1]

[1] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
[2] Institute of Informatics, University of Gdańsk, Poland
(maria.ganzha,marcin.paprzycki)@ibspan.waw.pl

Abstract. We are developing an agent-based intelligent middleware for the Grid (the *Agents in Grid* system). Resource providing agents are organized in teams and negotiate contracts with agents representing users. All information is ontologically demarcated and semantically processed. In particular, user preferences are turned into ontology class expressions and used directly in contract negotiations. The aim of this chapter is to discuss in details how ontologies are used in the *AiG* system. First, we briefly discuss the most important issues encountered in development of the *AiG* ontology. We follow, with the description of the implementation of ontology-focused parts of the system.

1 Introduction

The aim of this chapter is to discuss practical aspects of application of ontologies and semantic data processing in management of resources in the Grid. First, issues involved in development of an ontology of Grid computing are briefly considered. Discussed ontology is used not only to describe Grid resources, but also in Service Level Agreement (SLA) negotiations. Second, it is discussed how an ontology-driven user interface can be developed, to facilitate human-computer (i.e. human-software agent) communication. Third, a solution to the problem of ontology-based agent-agent communication is presented. Finally, the role of ontologies in SLA negotiations is outlined. The chapter begins with top-level description of the system, which will be used to illustrate these four main points.

The *Agents in Grid* (*AiG*) project aims at development of a flexible agent-based infrastructure, which is to facilitate intelligent resource management in the Grid. Thus, the project can be considered as an attempt at realizing the main idea underlining the seminal paper [5], where use of software agents as high-level middleware for the Grid was suggested. In the *AiG* project, it is proposed that flexible management of resources in the Grid can be provided by teams of software agents [10, 11]. Furthermore, the proposed approach is based on application of semantic data processing in all aspects of the system. Specifically, ontologies provide the metadata, to be used to describe resources, reason about them, and negotiate their usage. Finally, adaptability and flexibility of the system are to result from application of “agreement technologies,” agent negotiations, in particular.

2 System overview

In the work of Wasielewska et al, Grid is considered as an open environment (see, Chapter 7), in which *Agents* representing *User*s interact to, either (a) join a team, or (b) find team(s) to execute job(s) [1, 10]. The main assumptions behind the proposed approach were: [1]

- agents work in teams (groups of agents),
- each team has a single leader—the *LMaster* agent,
- each *LMaster* has a mirror, the *LMirror* agent that can take over its job,
- incoming workers (*Worker* agents) join teams based on *User*-defined criteria,
- teams (represented by their *LMaster*s) accept *Workers* based on team-specific criteria,
- each *Worker* agent can (if needed) play role of the *LMirror* or the *LMaster*,
- matchmaking is facilitated by the *CIC* component, represented by the *CIC Agent*.

These assumptions have been summarized in the Use Case diagram in Figure 1.

Let us now outline interactions between components of the system, i.e. the *User* and its representative, the *LAgent*, and agent teams represented by their leaders—*LMaster* agents (more information can be found in [2]). Since the system is based on semantic data processing, ontologies are used whenever applicable. Here, recall that utilization of ontologies as a method of knowledge representation, and basics of the OWL (Web Ontology Language) were introduced in Chapters 4 and 5, respectively. Let us now assume that the team "advertisements" describing: (1) what resources they offer, and / or (2) characteristics of workers they would like to "hire", are registered with the *Client Information Center* (*CIC*). Obviously, team advertisements are ontologically demarcated. Specifically, offered resources are represented with individuals (instances), and worker characteristics are represented with OWL class expressions. Let us focus on two main scenarios in the system: the *User* is looking for a team (1) to commission job execution, or (2) to join (to be paid for use of her resources). In both cases, the *User* interacts with her *LAgent* via an ontology-driven GUI application, and formulates conditions for (1) job execution, or (2) team joining. Respective descriptions of a desired resource(s) (or characteristics of (an) offered resource(s)) are also ontologically represented. Specifically, the GUI application allows the *User* to select such requirements on the basis of the existing *AiG ontology*, without any need of knowing it (see, Section 4). The resulting ontology class expression, is passed from the GUI to the *LAgent*. The *LAgent* communicates with the *CIC* (passes the ontology fragment to the *CIC Agent*; see, Section 5) to obtain a list of teams that satisfy the *User*-defined criteria. The *CIC* utilizes a reasoner to find individuals satisfying criteria found in the received ontology class expression. These individuals represent potential partner teams (their *LMasters*), and are send back to the *LAgent*. Next, the *LAgent* forwards the result to

[1] For a comprehensive discussion of reasons behind the approach, see [2, 14]

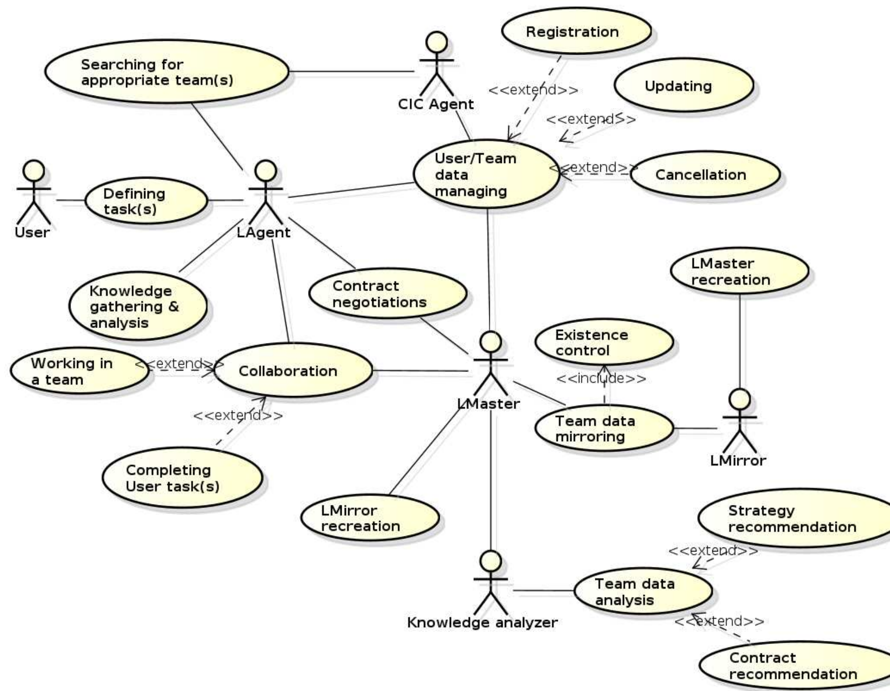

Fig. 1: Use case diagram of *AiG* system.

the GUI application and waits for the generated ontology fragment with *contract conditions* that the *User* specifies. Additionally, the *User* can limit the number of selected potential partner teams; based, for instance, on trust verification. Note that, similarly to the selection of required resource characteristics, specification of contract conditions is driven by the *AiG ontology* (its *contract ontology* part; see Section 6). Next, the *LAgent* communicates with the *LMasters* of selected teams, and they apply the *FIPA Iterated Contract-Net Protocol*[2] [14] to negotiate the contract (SLA) (see, Section 6). All information exchanged during the negotiations is based on the *AiG ontology*. The *LAgent* sends a *Call-For-Proposal* message that contains contract conditions, represented in the form of a class expression, and obtains (from the *LMaster*s) contract offers represented as ontology individuals. If the *LAgent* finds an appropriate team, a Service Level Agreement is formed. If no such team is found, the *LAgent* informs the *User* and awaits further instructions. Let us stress that this process applies to both, the job execution scenario and the team joining scenario. The only difference is in the details of negotiations (e.g. content of exchanged messages) taking place in each case.

[2] `www.fipa.org/specs/fipa00030/PC00030D.pdf`

3 Ontologies in the system

As stated above, when designing the system, it was assumed that *all* data processed in it will be ontologically demarcated. Therefore, after a brief reflection, it was realized that what was needed was an ontology, covering: (a) computer hardware and software (Grid resources), (b) Grid structure, (c) concepts related to the SLA and contract definitions. After a comprehensive investigation of existing Grid-related ontologies (see, [3]) it was decided to modify and extend the *Core Grid Ontology* (*CGO* [3]; [16]). While the *CGO* provided excellent base-terms concerning Grid resources and structure (parts (a) and (b)), there was a need to modify it slightly and to extend it to include the remaining concepts needed for the *AiG* system (concepts concerning part (c)). The complete description of the resulting ontology can be found in [3,4]. Here, let us briefly outline its main features. The extended *CGO* (the *AiG Ontology*) is structured into three layers (its core classes depicted in Figure 2):

1. *Grid Ontology*—directly extending the *CGO* concepts.
2. *Conditions Ontology*—includes classes required by the SLA negotiations (e.g. pricing, payment mechanisms, worker availability conditions, etc.); it imports the *Grid Ontology*, to use the terms related to the Grid structure and resources.
3. *Messaging Ontology*—contains definitions of messages exchanged by the agents, forming the communication protocols of the system (it uses the *Grid Ontology* and the *Conditions Ontology* to specify content of messages).

The crucial aspect of ontological modeling was the representation of constraints on ontology classes. For example, when a *User* is looking for a team to have a job executed, she needs to specify the necessary hardware (and possibly software) configuration. In this case, the common way of assigning values to class properties is not enough, as there is also a need to specify minimum, maximum, and range conditions. For instance, to execute her job, the *User* may need a processor that has at least 4 cores, but no more than 8 cores (these restrictions could be based on the knowledge of characteristics of the problem / job; e.g. parallelization methods used when it was implemented, and its parallel performance profile). After considering several approaches, to solve this problem, designers of the *AiG* system have settled on class expressions. Here, requirements are defined as a new class that restricts the set of individuals to these satisfying conditions on class properties. It is thus possible to ask a reasoner to infer a list of individuals of the generated class and receive these fulfilling the constraints.

While it may be possible to question some specific decisions made when completing the "re-design" of the CoreGrid ontology [4]—and interested readers are invited to send comments and suggestions to Wasielewska et. al., let us focus

[3] Unfortunately, the original CGO is not available online anymore and thus only the published work can be referenced

[4] its current version can be found at `http://gridagents.svn.sourceforge.net/viewvc/gridagents/trunk/ontology/AiGOntology/`

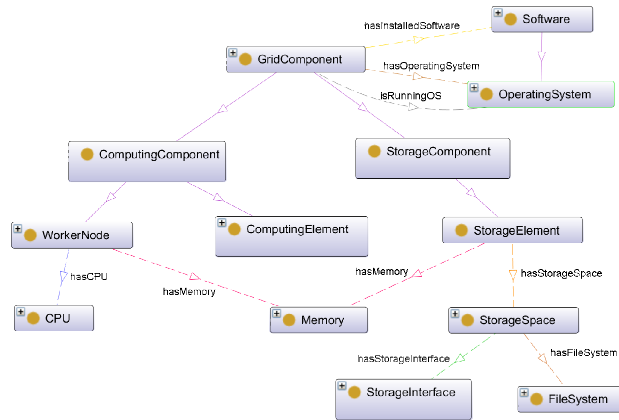

Fig. 2: Ontology diagram for *AiG* ontologies

on issues that arise when it is to be used in an actual application. Specifically, when it is to be the core data representation form in an agent-based system, and used to facilitate agent negotiations leading to an SLA. Note that, for space limitations, the focus of this chapter is on the "job execution scenario." However, all issues and results presented in what follows apply directly to the "team joining scenario."

4 Front-end design and implementation

The front-end developed for the system was designed to help the user communicate her needs and/or preferences using terms familiar and convenient for her, and then to translate her requirements into appropriate ontology fragments, e.g. into classes of the *AiG ontology*. In other words, the front-end subsystem, while becoming means to provide ontological data to the system (e.g. for the SLA negotiations), has to do this in a *User*-friendly way, without assuming the user's knowledge about semantic technologies and OWL specifically. After completing the requirements analysis, it was decided that, the front-end of the system has to consists of three main parts, allowing specification of requirements concerning:

1. *Scheduling job execution*—lets *User* specify hardware and software requirements that a team has to satisfy in order to be taken into consideration. In the second step of the scenario, the *User* creates a set of constraints on the contract for executing the job.
2. *Joining a team; Worker criteria*—specify information needed for negotiating joining a team, i.e. description of available resources. When an initial list of teams is found, the *User* also defines the restrictions on the contract between the *Worker* and the team.

3. *Joining a team; LMaster criteria*—also concerns worker joining a team. Here, the owner of the *LMaster* can specify conditions that must be met by any worker willing to join the team. These include, among others, hardware and software configuration of the Grid resource.

Note that, for brevity, material presented in this section is focused only on the first two sets of criteria (concerning direct *User*s of the system, rather than team managers). In this context, there exist two possible goals of the system. First, the system that would be 100% autonomous, where all decisions would be made by software agents, without further *User* participation (except of the initial specification of requirements). Second, *User* participation would be also possible / required / expected in specific stages of SLA negotiations. For instance, it would be possible for the *User* to manually filter the initial list of teams received from the *CIC*. Here, this could be considered as means of letting the *User* restrict the executors of her tasks only to the entities that she trusts (see, also [6]). While the first approach (total agent autonomy in representing *User*'s interests), can be seen as the "Holy Grail" of agent system design, it is the second approach that is more realistic (and has to be implementable; if not implemented). However, in the future, when reliance on autonomous software agents becomes a norm (e.g. when a complete system-wide trust management would be implemented), *User* involvement may not be needed (or, at least, considerably limited).

The design of the front-end of an agent system leads to a number of interesting problems. In the initial system prototype, the front-end was a desktop application with the *LAgent* running in the background (on the same machine). Advantages of that approach included simple architecture, and ease of interactions between the client application and the *LAgent*. Note however, that in this approach, a copy of the *AiG ontology* had to be stored locally (at least this would be the most natural solution). As a result any change in this ontology would have to be propagated to *all LAgent*s residing on *all User*-machines. Furthermore, this approach also meant that: (1) the *LAgent* could only work while the front-end application was running, and (2) at least a part of the *User*'s data was stored on the local machine. Therefore, meaningful interactions with the *LAgent* from different machines would be difficult (if not impossible). At the very least they would require installing the front-end software (including the ontology), on any such device. Since, currently, the possibility of accessing an application from *any* computer becomes highly desired (if not a necessity), Wasielewska et al. have decided to develop a web application that can be hosted in a "shared environment." Furthermore, such application, if properly designed, could help in solving the above mentioned problem of expected lack of *User* knowledge about ontologies. Finally, the proposed system could be friendly to potential ontology modifications. Therefore, it was decided to proceed with development of an ontology-driven front-end. Here, the vocabulary of specification of *User*-constraints would originate from the existing Grid ontology, hopefully simplifying tasks of *User*s of the system. Furthermore, the *AiG ontology* could be stored in a "single" place—with the application, considerably simplifying the

ontology maintenance. Note that the most natural place for the location of the application would be the *CIC* component (see, Figure 1).

Specify requirements for the team:

ComputingElement
http://www.owl-ontologies.com/unnamed.owl#ComputingElement

hasStorageSpace | is constrained by | StorageSpace | remove
hasFileSystem | is equal to individual | winnt | remove
hasTotalSize | is greater than | 200
and
hasOperatingSystem | is equal to individual | vista_sp2
Update
and

(a) Condition builder

The generated condition:


```
Response: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#" xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"> <owl:Class
rdf:about="http://gridagents.sourceforge.net/TeamConditions#TeamCondition"> <owl:intersectionOf rdf:parseType="Collection">
<rdf:Description rdf:about="http://www.owl-ontologies.com/unnamed.owl#ComputingElement"/> <owl:Restriction>
<owl:someValuesFrom> <owl:Class> <owl:intersectionOf rdf:parseType="Collection"> <rdf:Description rdf:about="http://www.owl-
ontologies.com/unnamed.owl#StorageSpace"/> <owl:Restriction> <owl:hasValue rdf:resource="http://gridagents.sourceforge.net
/AiGGridOntology#winnt"/> <owl:onProperty rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasFileSystem"/>
</owl:Restriction> <owl:Restriction> <owl:hasValue>200</owl:hasValue> <owl:onProperty
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasTotalSize"/> </owl:Restriction> </owl:intersectionOf>
</owl:Class> </owl:someValuesFrom> <owl:onProperty rdf:resource="http://gridagents.sourceforge.net
/AiGGridOntology#hasStorageSpace"/> </owl:Restriction> <owl:Restriction> <owl:hasValue
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#vista_sp2"/> <owl:onProperty
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasOperatingSystem"/> </owl:Restriction> </owl:intersectionOf>
</owl:Class> </rdf:RDF>
```


(b) OWL class

Fig. 3: A condition builder section

The core of the front-end is a condition builder—a set of condition boxes, each representing a description or constraint on a single class-property relationship (see, Figure 3 for screen-shots from the running front-end, representing the condition selection process, and the resulting OWL class). Depending on the selected class, the *User* may choose one of properties that the class is in the domain of. For instance, having selected the *WorkerNode* class, the expanded property box will contain properties such as *hasStorageSpace*, *hasMemory* or *hasCPU*.

Next, she can specify an *operator*, from the set of applicable ones, to the selected property. For example, for the datatype properties these may include: *equal to*, or *greater than*, *less than* whereas for object properties these would be *is equal to individual* and *is constrained by*. When an operator is selected, the system generates a "new" fragment of the user interface, used to specify value of the property. Again, controls depend on the selected operator—be it a simple text box, or a drop down list of applicable individuals. It is an important

feature of this component that both: available properties, and possible class arguments, are inferred directly from the ontology using a reasoner, which means that the application fully supports class and property inheritance and other, more complex relations between the elements of the ontology.

To illustrate the relationship between the ontological metadata and the structure of user interface elements, let us look at the following examples. In Figure 4 a drop-down list of properties that can be applied to the selected class – the *PhysicalMemory* – is presented. The elements of this list are generated from the ontology, the relevant part of which is contained in the following (RDF/XML) snippet [5]. Notice that these properties are actually defined in two different ontologies (listed in a single snippet, for clarity and brevity) and, furthermore, they do not specify the *PhysicalMemory* directly in their domain. This shows how the usage of a reasoner, when analyzing the metadata of the ontology, can help in making the system more robust and flexible.

```
ObjectProperty: belongToVO
 Domain: GridEntity
 Range: VO
ObjectProperty: hasID
 Domain: GridEntity
 Range: URI
DataProperty: hasTotalSize
 Characteristics: Functional
 Domain: Memory or StorageSpace
 Range: int
DataProperty: hasAvailableSize
 Domain: Memory or StorageSpace
 Range: int
DataProperty: hasName
 Domain: GridApplication or GridEntity
 Range: string
```

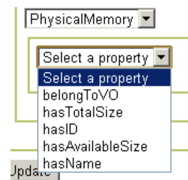

Fig. 4: Selecting a class property

In the next example, it is demonstrated how the *User* can specify that a property should be *equal to* a specific individual contained in the ontology. Figure 5 shows a list of individuals that can be used as a value of the property *hasArchitecture* for class *WorkerNode*. These reflect the following individuals from the ontology:

[5] All ontological snippets, cited in the text, shall be presented in the Manchester OWL Syntax with namespaces omitted for readability and space preservation.

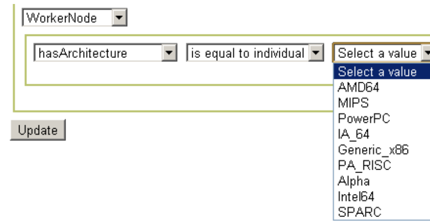

Fig. 5: Selecting an individual

Turning the attention to more complex use cases, an interesting one is that of nested constraints. For object properties, when the *User* selects the operator *is constrained by*, for a class to be further specified, a new condition box is created within the existing one. It is used to describe the details, or requirements, regarding the value of the selected property. The front-end supports also setting constraints on multiple properties of the same class, using the *and* buttons, which add a new condition box at the same level as the previous one.

As an example let us consider a *User* specifying that the resource required for running her job should have a multi-core processor with clock speed greater than 1.4 GHz. This can be easily specified in the application as shown on Figure 6. The *User* first specifies that the computing element should have the value of the *hasWN* set to an instance of the *WorkerNode* class. This instance is in turn constrained to an individual with an instance of the *CPU* class, as the value of the *hasCPU* class. Finally, the two conditions are set on the properties of the *CPU* class: the *hasCores* (greater than one) and the *hasClockSpeed* (greater than 1400 MHz). The result of such specification, translated into the OWL by the server component is shown in the following listing.

```
Class: TeamCondition
 EquivalentTo: ComputingElement that hasWN some (WorkerNode that
hasCPU some (CPU that hasClockSpeed some integer[> 1400] and hasCores
some integer[> 1]))
```

When the *User* finishes specifying conditions and pushes the *submit* button, the system parses the internal representation of the conditions, and transforms it into an *OWL Class Expression*. This OWL fragment is passed to the *JADE GatewayAgent*, responsible for passing information between the application server, and the *JADE* agent container. The *GatewayAgent* forwards the data to the *LAgent*, to handle it within the system.

Here, it is worthy stressing (again) that in the front-end, all elements from which the *User* builds the ontological conditions and descriptions are generated dynamically, from the structure of the ontology. Therefore, all changes to the ontology can be applied automatically during the system runtime. This is extremely important, especially in the case of ontology matching and enriching, based on the information received from other agents. It also simplifies, in one more way, maintenance of changes in the ontology. For instance, if a new class

Fig. 6: Example of a class constraint

of NVidia processors will be introduced, the necessary changes in the ontology will almost automatically materialize in the front-end.

Furthermore, user interface elements are built dynamically, in response to *User* actions. For example, if the *User* wishes to specify a particular CPU architecture, individuals of the *CPUArchitecture* class will only be fetched from the ontology when the *User* selects an *equal to individual* condition on the *hasArchitecture* property. This allows the processing to be limited only to the needed parts of the ontology. Moreover, it allows to base displayed options on the *User*'s previous choices. Observe that this could be the basis of developing a mechanism for providing automated assistance to the *User*, by suggesting the most useful or common options, or by filtering out inconsistent options.

The part of the user interface responsible for defining the concrete instances (e.g. the hardware and software configuration of a particular Grid resource), is built around the same condition builder components. Of course, here the available property operators are restricted to the *equal to* and the ontology elements generated by the OWL generator represent individuals instead of class expressions. Moreover, the class condition constraints have been modified slightly (for better *User* experience); the *is constrained by* operator has been replaced with the *is described with*. The functionality of specifying descriptions of individuals, instead of class expressions, is used, among others, when defining the hardware and software configuration of a resource that is offered to (join) a team.

The example, displayed in Figure 7, illustrates a description of a *WorkerNode* having total size of 1500 MB of storage space formatted using the *ext3* file system. The following snippet shows the rendering of the individual representing such resource as returned by the OWL generator.

```
Individual: WorkerDescription
 Types: WorkerNode
 Facts: hasStorageSpace _:storage
Individual: _:storage
 Types: StorageSpace
 Facts: hasFileSystem ext3, hasTotalSize 1500
```

Fig. 7: Example of a class constraint

Another interesting challenge that was encountered, while migrating the GUI from a desktop based application towards a web-based one, was that of passing messages between the web controllers and the agents. Although the *JADE* agent environment contains special classes that provide valuable help in such scenarios (the *Gateway* and the *GatewayAgent* classes), and makes sending messages from a non-agent environment rather straightforward, handling requests coming from agents, within the user interface, is less trivial (for an interesting discussion and another possible solution, see [7, 8]). This is mostly due to the fact that web applications are stateless by nature, and therefore it is not directly possible to implement an event-based system where a message coming from an agent triggers a particular action in the GUI. Instead, it is necessary to implement a queue of messages received by the *GatewayAgent*, representing the user interface, and some form of a polling mechanism that would check for new messages. In our implementation, the *GatewayAgent* is responsible for keeping a list of conversations containing messages. Through the use of the *Gateway* class, the web controllers are able to reach the message queue. Polling itself is achieved by using the client side *AJAX* requests.

The front-end application has been developed on top of the *Play! Framework* [6]—a lightweight web framework offering straightforward deployment and a rich plugin ecosystem. This framework also serves as the technological stack for the server part of the ontology builder user interface, which comprises the web controllers as well as modules for reading ontological metadata and generating the OWL data from the descriptions provided by the *User*. The browser-side subsystem is implemented as a dynamic *JavaScript* application using *jQuery* [7]—one of the most popular general purpose *JavaScript* libraries. The remaining part of the application—the *JADE Gateway* component is created as a standalone Java library, exposing an API for initiating agent conversations, sending messages and retrieving contents of a specific conversation. After some additional testing,the *JADE Gateway* and the ontology builder user interface are to be re-

[6] www.playframework.org
[7] www.playframework.org

leased as *Play!* modules, to be easily integrated into other *Play!* applications. Furthermore, the *JADE Gateway* will be released as a *JADE* add-on. It is worth noting that the *AJAX* functionality for listening on agent's responses has been developed as a *jQuery* plugin, enabling its easy embedding into any HTML page. Currently, the plugin is developed using a simple polling mechanism controlled on the browser-side with a request sent to the server every specific number of seconds. In the future versions of the software this mechanism willnbe replaced with a less server-consuming implementation based on the *Comet / Long polling mechanism* [8]

5 Passing ontological information; integrating front-end and back-end

Let us now assume that, as described above, the *User* requirements / constraints have been specified and transformed by the user interface into OWL class expressions / individuals and passed to the *LAgent* representing the *User*. Next, such information has to be passed further to various components of the system. For instance, it could be passed to the *CIC* infrastructure (the *CIC Agent*) to query for agent teams satisfying *User*'s needs. It can be also sent to the *LMaster* agents as a part of SLA negotiations. As noted, all these processes involve ontological matchmaking (which was introduced in chapters 6 and 7). In summary, communication in the *AiG* system relies on passing around, extracting information from, and manipulating instances of, ontologies. However, issues raised here apply to any agent based system that is to use in practice ontologies and pass their "fragments" around for semantic data processing. Without a flexible and robust support, use of ontologies in agent systems (e.g. as envisioned in the classic paper by J. Hendler [9]) will be overly complex, thus reducing their uptake.

Unfortunately, the default *JADE* ontological facilities are very limiting. In this framework, ontologies are stored in *static Java classes*. Those classes are not shared between agents (i.e. each agent needs to have a private copy) and, when used for communication purposes, may lead to misunderstandings between agents. The default *JADE* codecs for ontological communication can encode the Java classes into the FIPA SL (FIPA Semantic Language [9])—a language used to describe the context of any *JADE* ACL message. Using the FIPA SL in both context and content of the message is disadvantageous, as there is currently no good reasoner for this language. As a matter of fact, it seems that there is currently no publicly available FIPA SL reasoner. Moreover, the FIPA SL is not decidable, which may sometimes prevent an agent from "understanding" the content of the FIPA SL-encoded message. Using a non-decidable ontology language is simply not possible in the *AiG* system, because the problem domain requires introducing new, as well as changing the already existing data. Under

[8] Comet and Reverse Ajax: The Next-Generation Ajax 2.0", `http://dl.acm.org/citation.cfm?id=1453096`

[9] `http://www.fipa.org/specs/fipa00008/SC00008I.html`

such conditions it would be impossible to guarantee decidability of ontology at any time. Managing an ontology that is not decidable would require writing new reasoning algorithms capable of dealing with undecidability. Using an ontology language that is decidable is a much simpler and more feasible solution.

It should be stressed that having the ontology "constrained" within static classes means that there is no practical way to quickly add new class expressions, properties or constraints to the ontology. Any change in the ontology would require change in the Java files for *every* agent. New files would need to be sent to every agent and swapped with the old ones (possibly via dynamic class loading). In any case, updating *JADE* ontologies is extremely impractical and requires reloading, which in turn means that, for all practical purposes, the system would need to stop working during the ontology update process. The Wasielewska et al.'s solution, outlined here, does not suffer from such penalties.

Observe also that, from the practical point of view, *JADE* ontologies are very hard to manage and do not offer many of the useful features that are present in the OWL 2.0. For instance, there is no multiple inheritance (which is also a property of Java itself), there are no cardinality restrictions, or datatype facets and reasoner support is missing. Using only the *JADE* ontologies, there is no way to, for example, define a team condition restricted to having exactly two computers with between 4 and 8 processor cores. Creating such a class expression would require writing custom Java code to supplement the *JADE* classes. As a result it is not possible to create the team condition dynamically and contain its entire description within a *JADE* ontology class. One of the biggest downsides of *JADE* ontologies is also that they are hardly reusable. They cannot be used outside of a *JADE* agent system, which makes them rather unpopular. All these disadvantages of the *JADE* ontologies make them applicable only to very simple ontologies with basic vocabularies. Let us now present a solution that does not suffer from such problems.

As presented above, in the *AiG* system, it is essential to be able to transfer arbitrary fragments of OWL ontologies, including TBox definitions of classes, used for representing constraints and requirements. Previously, this problem has been discussed in [13], and resulted in creation of the *JadeOWL Codec* [12]. Unfortunately, this plugin was extremely tightly integrated with the commercial RacerPro [10] reasoner, and its development seems to have stopped before the release of the OWL 2.0 specification. Therefore, a *JADE* plugin called *JadeOWL* was developed, aiming at providing OWL support to the agent message processing. The *JadeOWL* uses the OWL API [11] interface and improves upon it by integrating it with *JADE* communication routines and adding other useful features.

Direct mapping of the OWL 2.0 [12] into any static object-oriented programming language is not possible; i.e. there is no way to represent OWL as Java classes while preserving its dynamic structure and properties (partial solution to this problem can be found in [15]). Therefore, as opposed to the existing so-

[10] `http://www.racer-systems.com/`
[11] `http://owlapi.sourceforge.net/`
[12] `http://www.w3.org/TR/owl2-overview/`

lutions, it was decided that any information instance, such as information about teams or negotiation deals, will be stored and accessed as OWL formatted text files. Thus, the plugin had to provide interface to files viewed both as a raw text, and as an OWL ontology; i.e. after passing a raw file, the plugin had to be able to probe the structure of the ontology, extract classes and instances, as well as their properties. In this way the plugin had to be able to serve as a high level interface to the structure and content of ontological messages, passed between *JADE* agents.

In communication scenarios considered here, data is prepared as a piece of the OWL ontology. The OWL content that is encoded in the message can contain any valid OWL entity including classes, instances, properties definitions, annotation properties, imports declarations and so on. The actual syntax can be any OWL expression supported by the OWL 2.0. Currently the supported syntaxes include: Manchester, functional, RDF/XML, OWL/XML, and the Turtle syntax. Note that, as indicated above, the *AiG* ontologies are stored and communicated in the RDF/XML format. Using this format guarantees that the used ontologies can be read by any OWL 2.0 tool because the support for the RDF/XML in the OWL ontologies is a requirement set by the official OWL documents.

Although messages contain raw OWL data, their interpretation is done internally by the plugin, which separates the syntax from the semantics. In this way agents can access the information without the need to parse the text. This interpretation requires reasoning about the data. Therefore, an instance of a semantic reasoner had to be bundled with the communication plugin. *JadeOWL* currently supports: HermiT [13], Pellet[14], and Fact++[15] reasoners. However, in the *AiG* implementation, the Pellet reasoner is used. Note that reasoners are used not only by the codec, but also provide the infrastructure for all agent reasoning. For instance, the *CIC* infrastructure uses it to match registered resource descriptions (teams, or worker candidate, profiles) against restrictions expressed with class expressions—for instance, in the two scenarios described in Section 2.

The design of the system requires agents to have shared (public) knowledge, as well as private knowledge. For example every agent in the Grid needs to understand basic concept such as a computing node. This knowledge is considered to be shared by every agent and does not depend on its role in the system. On the other hand, detailed information about every team in the Grid is, by design, gathered in the *CIC* infrastructure. This information can be considered an example of the private knowledge of the *CIC*; i.e. it is up to the *CIC Agent* to decide how and with whom to share this knowledge. The separation of knowledge into public and private parts creates a need for a query language that would allow agents to ask specific questions about the private knowledge of other agents. This need is satisfied by the *JadeOWL* A-Box query language.

The *JadeOWL* query language provides a way to ask questions about the OWL ontologies using pieces of the OWL code. Any query can be answered

[13] HermiT OWL Reasoner, `http://hermit-reasoner.com/`
[14] Pellet: OWL 2 Reasoner for Java,`http://clarkparsia.com/pellet/`
[15] OWL: FaCT++”,`http://owl.man.ac.uk/factplusplus/`

locally or sent to another agent to be answered there. An answer to a query is a piece of data in the OWL format. To extract data (e.g. an OWL instance) from an ontology, a custom OWL class is created—a defined class called the "query class." The exact structure of this class depends on the data that needs to be extracted. For example, if the *CIC* is asked for agent teams with an IBM Linux machine, it sends information received from the *LAgent* to the *JadeOWL* plugin. The plugin creates an OWL class that extends the definition of the OWL class describing the team advertisements, but also contains an OWL property restrictions that forces any instance of this class to be a team with an IBM Linux computer. Other types of restrictions (like the cardinality restriction) supported by the OWL 2.0 [16] are also available.

Here, the reasoner performs consistency and satisfiability tests on the new class in the context of the ontology. If the tests fail, it means that either the class cannot have any instances or it contradicts other information in the ontology. In this case, an exception is thrown and the reasoner output is routed back to the creator of the instance, to inform about the problem and, possibly, how to fix it. After passing the tests, the class prepared in this way is presented to the reasoner that finds all its instances. The prepared OWL instances are sent back to the *LAgent* that requested the information.

The *JadeOWL* is used in any communication routine required by the system. For example, advertising a team by the *LMaster* involves sending an instance of an OWL class (describing the team) to the *CIC*, which recognizes it as a team advertisement and stores it in an OWL file. When asked by the *LAgent*, it filters all stored instances, to satisfy the specified constraints.

Summarizing, the *JadeOWL* plugin aids creation of OWL classes and instances by producing and structuring the actual OWL text, while the reasoner (that is internal to the plugin) performs the validity/consistency checks and filtering. The A-Box query system assists in finding teams or agents that fit the criteria defined by the *User* via the GUI. The *JadeOWL* also intermediates in the agent-to-agent communication, and makes full ontological communication available, while preserving constraints set upon ontologies in the OWL format. Finally, it makes it possible to exploit the dynamic nature of the OWL.

6 Negotiations in the system

Let us now assume that the preliminary processes (in the scenarios described in Section 2) been completed and a group of team managers (*LMaster* agents) has been selected as potential job executors (see, 8). In the next step, the SLA negotiations ensue. The SLA, defines agreement reached by the parties, while negotiations are understood as a flow of messages between "parties" (in this case the *LAgents* and the *LMasters*). It should be obvious that, since in the *AiG* system, ontology fragments are passed as the message content (using the above described codec), all negotiation parameters and contract conditions are

[16] `http://www.w3.org/TR/owl2-overview/`

represented with respective class expressions and properties (from the *AiG ontology*). As stated in [14], the negotiation process is based on the *FIPA Iterated Contract-Net Protocol*, and involves both *negotiable* e.g. deadline penalty, job execution timeline, and *static* parameters e.g. resource description specified by the *User* through the front-end (described in Section 4). Currently, a simplified version of the protocol e.g. the *FIPA Contract-Net Protocol* is utilized, however, in the future its complete multi-round version of will be utilized.

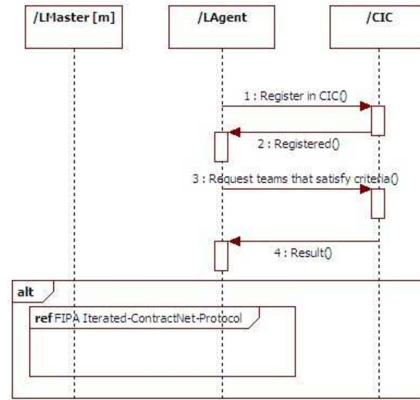

Fig. 8: Sequence diagram for job execution scenario

After *User* specifies contract conditions and restrictions, an appropriate ontology with class expression is generated (by the GUI, see Section 4) and send to the *LAgent*, which constructs a *Call-For-Proposal* message with an OWL class expression representing restrictions on contract conditions (for either one of the negotiation scenarios) including also the required resource description—for the job execution scenario; or of a resource that the *User* wants to sell—for the team joining scenario. This message is sent to the selected *LMasters*, and those interested in the proposal reply with the OWL instances—individuals representing their offers. Before replying, each *LMaster* agent assesses received offers, based on its internal criteria e.g. checking if any team member suitable to do a job is available. The *LAgent* verifies if received contract offers match it's criteria and selects the best offer, in the case that one can be selected. In Figure 9 the sequence of messages exchanged during the negotiation process based on the *FIPA Contract-Net Protocol* is depicted.

The following snippet shows a simple class expression with restrictions on the contract, where the deadline penalty should be less than 100, fixed resource utilization price should be less than 500, and the required resource should run Windows Vista SP2 operating system.

```
ObjectProperty: contractedResource
```

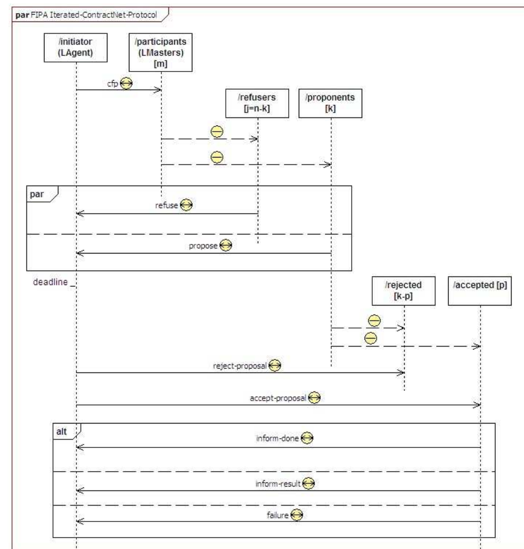

Fig. 9: Sequence diagram for contract negotiations

```
ObjectProperty: fixedUtilizationPrice
ObjectProperty: paymentConditions
ObjectProperty: isRunningOS
DataProperty: deadlinePenalty
DataProperty: peakTimePrice
Class: JobExecutionConditions
Class: PaymentConditions
Class: Pricing
Class: WorkerNode
Individual: vista_sp2
Class: JobExecutionConditions
 EquivalentTo: JobExecutionConditions that contractedResources some
(WorkerNode that isRunningOS value vista_sp2) and paymentConditions some
(PaymentConditions that fixedUnitizationPrice some (Pricing that
peakTimePrice some float[<= 500])) and deadlinePenalty some float[< 100]
```

In response to such a *CFP* message, the following snippet shows a potential offer (contract proposal) instance generated by the *LMaster* agent. Presented contract offer specifies the deadline penalty to be 91.56, and a fixed utilization price to be 450.0. Obviously, in both cases of the *CFP* and the contract proposal, the prices are represented in some imaginary currency.

```
Individual: Contract
 Types: JobExecutionConditions
 Facts: deadlinePenalty "91.56"^^xsd:float, paymentConditions
ContractPaymentConditions
Individual: ContractPaymentConditions
 Types: PaymentConditions
 Facts: fixedUtilizationPrice "450.0"^^xsd:float
```

Note that, reasoning in the back-end part of the system is required for both negotiating parties, i.e. the *LAgent* and the *LMaster*s in order to select best of-

fer, or respectively verify if an offer can be prepared (e.g. the contract conditions are acceptable). In the initial proof of concept application, the *LAgent* utilized a linear-additive model for three predefined criteria to select an offer [10]. This model is a simple MCA model, in which criteria are treated and assessed independently. In the future, both parties shall use multicriterial analysis to evaluate received proposals and make offers that take into consideration their own ability to fulfill required conditions, as well as preferences. Additional criteria, that became available in the ontology (as compared to the proof of concept application) are going to be considered. Both negotiation parties should be able to handle arbitrary constraints from the *AiG ontology* based on the restricted property datatype and weight.

Upon receiving the offer from the *LMaster* agent, the *LAgent* uses the reasoner to verify if the received contract offer satisfies the criteria provided by the *User*. On the other hand, the *LMaster* agents may use reasoning and MCA to determine, for instance, the cost of job execution. Note that each resource needed for job execution, e.g. memory, I/O bandwidth, has a pricing property in the ontology. This property specifies the pricing type and the price. To evaluate the total price of the job execution, the *LMaster* combines prices for each required component. Reasoning is also used by the *LMasters* to verify if they are able to execute a given job i.e. if there is an available member in the team that has resources required to execute a specific job. So far, team members resource descriptions have been stored in the *CIC* component, however, they will be stored also locally so that the *LMaster* can use reasoner on it's local ontological database.

7 Concluding remarks

The aim of this chapter was to illustrate how ontologies and semantic data processing can be used in an actual application, to facilitate contract negotiations. It describes in detail the front-end, which allows use of the application without knowledge of ontologies, the front-end back-end integration that allows agent systems use ontologies without the need to turn them to Java classes, and the initial design of the negotiation mechanism. Since all the needed "front-end-tools" are in place, the focus of research will now shift to the extending the existing simple SLA negotiations.

Acknowledgements

Work of Maria Ganzha and Michał Drozdowicz was supported from 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).

References

1. Mariusz Dominiak, Maria Ganzha, Maciej Gawinecki, Wojciech Kuranowski, Marcin Paprzycki, Svetozar Margenov, and Ivan Lirkov. Utilizing agent teams

in grid resource brokering. *International Transactions on Systems Science and Applications*, 3(4):296–306, 2008.
2. Mateusz Dominiak, Wojciech Kuranowski, Maciej Gawinecki, Maria Ganzha, and Marcin Paprzycki. Utilizing agent teams in Grid resource management—preliminary considerations. In *Proc. of the IEEE John Vincent Atanasoff Conference*, pages 46–51, Los Alamitos, CA, 2006. IEEE CS Press.
3. Michał Drozdowicz, Maria Ganzha, Marcin Paprzycki, Richard Olejnik, Ivan Lirkov, Pavel Telegin, and Mehrdad Senobari. Parallel, distributed and grid computing for engineering. chapter Ontologies, Agents and the Grid: An Overview, pages 117–140. Saxe-Coburg Publications, Stirlingshire, UK, 2009.
4. Michał Drozdowicz, Katarzyna Wasielewska, Maria Ganzha, Marcin Paprzycki, Naoual Attaui, Ivan Lirkov, Richard Olejnik, Dana Petcu, and Costin Badica. Trends in parallel, distributed, grid and cloud computing for engineering. chapter Ontology for Contract Negotiations in Agent-based Grid Resource Management System. Saxe-Coburg Publications, Stirlingshire, UK, 2011.
5. Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. *Autonomous Agents and Multiagent Systems, International Joint Conference on*, 1:8–15, 2004.
6. Maria Ganzha, Marcin Paprzycki, and Ivan 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*, volume 946 of *AIP Conf. Proc.*, pages 35–46, College Park, MD, 2007. American Institute of Physics.
7. Maciej Gawinecki, Minor Gordon, Paweł Kaczmarek, and Marcin Paprzycki. The problem of agent-client communication on the internet. In *Scalable Computing: Practice and Experience*, volume 6(1), pages 111–123, 2005.
8. Minor Gordon, Marcin Paprzycki, and Violetta Galant. Agent-client interaction in a Web-based E-commerce system. In Dan Grigoras, editor, *Proceedings of the International Symposium on Parallel and Distributed Computing*, pages 1–10, Ias,i, Romania, 2002. University of Ias,i Press.
9. James Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37.
10. Wojciech Kuranowski, Maria Ganzha, Maciej Gawinecki, Marcin Paprzycki, Ivan Lirkov, and Svetozar 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.
11. Wojciech Kuranowski, Maria Ganzha, Marcin Paprzycki, and Ivan Lirkov. Supervising agent team an agent-based grid resource brokering system—initial solution. In Fatos Xhafa and L. Barolli, editors, *Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems*, pages 321–326, Los Alamitos, CA, 2008. IEEE CS Press.
12. Bernhard Schiemann. JADEOWL Codec. `http://www8.informatik.uni-erlangen.de/en/demosdownloads.html`.
13. Bernhard Schiemann and Ulf Schreiber. OWL DL as a FIPA ACL content language, 2006.
14. Katarzyna Wasielewska, Michał Drozdowicz, Maria Ganzha, Marcin Paprzycki, Naoual Attaui, Dana Petcu, Costin Badica, Richard Olejnik, and Ivan Lirkov. Trends in Parallel, Distributed, Grid and Cloud Computing for engineering. chapter Negotiations in an Agent-based Grid Resource Brokering Systems. Saxe-Coburg Publications, Stirlingshire, UK, 2011.

15. Y. Xin-yu and L. Juan-zi. Research on Mapping OWL Ontology to Software Code Model. 2009.
16. Wei Xing, Marios D. Dikaiakos, Rizos Sakellariou, Salvatore Orlando, and Domenico Laforenza. Design and Development of a Core Grid Ontology. In *Proc. of the CoreGRID Workshop: Integrated research in Grid Computing*, pages 21–31, 2005.