Utilization of Modified CoreGRID Ontology in an Agent-based Grid Resource Management System

M. Drozdowicz[(1)], M. Ganzha[(1,2)],
K. Wasielewska[(1)], M. Paprzycki[(1,3)],
[(1)] SRI PAS, Warsaw, Poland,
[(2)] University of Gdańsk, Poland
[(3)] Warsaw Management Academy,
Poland,

I. Lirkov[(4)], R.Olejnik[(5)],
N. Attaoui[(6)]
[(4)] IPP BAS, Sofia, Bulgaria
[(5)] CNRS, Lille,France,
[(6)]Abdelmalek Essaadi University,
Tetouan, Morocco

Abstract

The *Agents in Grid* project is devoted to the development of an agent-based intelligent high-level Grid middleware. In the proposed system, all data processing is ontology-driven, and initially was based on an in-house developed mini-ontology of the Grid. Our recent analysis has indicated that we should adapt and utilize the Grid ontology developed within the framework of the CoreGRID project. This note outlines how we have modified and extended the CoreGRID ontology to fulfill the needs of our approach.

1 Introduction

In our work we are following the main idea put forward in [8], where it was suggested that software agents can infuse computational Grids with the needed intelligence. Furthermore, agreeing with J. Hendler, who in [9] argued that software agents and ontologies should work very well together, we are developing the system based not only on software agents, but also on utilization of ontologically demarcated data and semantic reasoning. Finally, accepting arguments stated in [12], we believe that the only way to effectively address objections raised against projects involving software agents is by implementing such systems and studying their properties; such as usability / efficiency / robustness / flexibility, etc.

With these three fundamental assumptions guiding our work, in this note, we focus our attention on the ontology that is to be used in the *Agents in Grid* (*AiG*) system. In our recent work ([7]), we have analyzed the existing Grid resource description languages, ontologies developed for computational Grids, and ontologies that targeted systems involving agents and Grids. Our analysis resulted in the decision to use, as the basis for our system, ontology developed within the framework of the CoreGRID project ([13]). Unfortunately, we have found that even this ontology, while the closest to fulfilling our needs, will have to be modified and extended first. The aim of this contribution is to: (a) outline the way that ontologies are used in our system, (b) summarize main issues that had to be resolved when adapting the CoreGRID ontology, and (c) sketch main facets of the proposed solution. This is also how this note is structured.

2 Ontologies in the system

Due to the space limitations, readers interested in the detailed description of our project are referred to [6, 10, 11]. In the proposed system, software agents work in teams managed by team leaders (the *LMaster* agents). There are two main processes (scenarios) taking place in the system. First, when the *User*, represented by its agent (the *LAgent*), would like to contract job execution. Here, after job execution *constraints* are specified (using a web-based interface), a SPARQL query ([5]) describing needed resources is formulated and answered by the *Client Information Center* (*CIC*; central repository of information about all agent teams available in the system, their resources, and sought team members). In the next step, the FIPA Contract Net Protocol (CNP; [1]) is applied, either to find a team to complete the task, or to report to the *User*, that a team satisfying the *User*-defined conditions cannot be found. Second, when the *User* would like to have her computer(s) work within a team (and earn money). In this case the scenario is "structurally" almost the same. Specification of team-joining *constraints* results in interactions between the *LAgent* and the *CIC*, followed by the FIPA CNP-based negotiations, resulting in team selection or in failure to find one (reported to the *User*).

It should be stressed that *all* information stored and processed in the system consists of ontologically

demarcated data. Specifically, in the context of the above outlined scenarios, we can see: (1) team data (available resources, and information about sought members) is stored as a persistent model in the Jena repository [3], using either the *TDB* subsystem with custom storage and indexing, or the *SDB* subsystem based on the SQL database[1], (2) querying of the *CIC*-held repository involves SPARQL queries, (3) results of such SPARQL queries are: (i) lists of teams that meet job execution constraints, or (ii) lists of teams that can be joined, described as instances of the *TeamAd* class (id of the team *LMaster* and resources offered by the team), (4) all communication between agents utilizes the FIPA ACL (Agent Communication Language) messages with content interpreted according to the system-defined ontologies, (5) FIPA Contract Net Protocol calls for proposals that contain *User*-specified constraints, and responses that involve proposed job execution details, or team-work contract specifications are instances of ontology classes.

Note that we omit various aspects of communication between the *LAgent*, or the *LMaster*, and the *CIC*, as they do not address any additional facets of ontology-based data processing. Instead they should be approached from the perspective of design and implementation of distributed agent-based systems.

3 Limitations of *CoreGRID* ontology

Let us now consider the question of ontology to be used in the proposed system. While initially ([6, 10, 11]) we have used an in-house developed minimalistic ontology, we well were aware of the fact that a feature-complete, robust ontology of the Grid will be needed. Simply said, we wanted first to try some ideas, before starting to utilize a large-scale ontology. Therefore, only recently, we have performed a comprehensive evaluation of existing (explicit and implicit) ontologies appearing in context of the Grid, and ontologies developed for projects attempting at fusing agents and Grids (see, [7] for a complete description of results of our investigations). The main outcome of this work was the decision to utilize the CoreGRID Ontology (*CGO*, [13]). While this ontology was the closest to our needs, nevertheless, (1) we found a number of places where realizations of certain concepts did not fit what we perceive as needed for a complete Grid resource description, and (2) the *CGO* did not cover requirements related to contract negotiations and therefore had to be extended.

To illustrate our concerns, let us look at sample scenarios that need to be supported through ontologically demarcated data, but were not handled well by the CoreGRID Ontology in its original form.

[1]The *TDB*, and the *SDB*; are Jena RDF storage components, which support the SPARQL query engine.

3.1 Memory requirements

Let us consider a case when the *LAgent* is looking for a team to execute a job. This job is based on an implemented algorithm, which has well-defined memory-related constraints. For instance, it needs a large amount of directly available fast-access RAM, and extra virtual (or swap) memory for less frequently used data. In the *CGO* the only property describing the amount of available memory of the resource was the *mainMemory* property, which is clearly insufficient to support the aforementioned scenario.

Another aspect of memory-related properties that was important to us was the specification of the domain and the range of properties. In our application we aim at creating a web-based user interface (UI) that lets the *User* (interactively) specify values of properties defined in the ontology. To build such UI we need discoverability of properties that can be used to describe the instance of a class (using the domain of the property) and a set of possible values that the *User* can input (using the range of the property). Unfortunately, definitions of properties describing the resource memory in the CGO (*RAMSize* and *mainMemory*) did not contain the specification of either the domain or the range and therefore could not be used in this context (as originally defined).

3.2 Storage requirements

The original CGO lacked properties that could be used to describe the storage elements of the resource. Specifically, while some classes representing these concepts were available (*StorageSpace*, *StorageInterface*, *FileSystem*), at the same time there were no properties that linked them to other classes. Due to the fact that we need to describe the Grid resource using these concepts and that our UI requirements necessitate the existence of strongly typed properties linking the classes together (see previous section for more details) there was a clear need to introduce them.

3.3 Software requirements

The CoreGRID ontology contained classes and properties to describe the software configuration of a Grid resource, e.g. subclasses of the *Software* class (*ApplicationEnv*, *Exec*, *Lib* and *OperatingSystem*), and the *installedSoftware* property. These concepts were sufficient for the utilization of the ontology in scientific Grid scenarios, however, in our application we

need to support more involved scenarios. The heterogeneity and globality of the Grid, as we see it, means that there might be cases of resources containing more than a single operating system. These might be either computers having multiple operating systems installed as multiboot partitions or simply hosting different virtual machines (so called, server virtualization). In such case we should differentiate between having an OS installed (and potentially available), and an OS actually running at a given time. While the importance of the information stating which OS is currently running on a given node is rather obvious, the *LMaster* might be also interested in knowing what other OS's the *Worker* is capable of running at some other time (this information may be usable, for instance, during evaluation of a team joining offer).

Another issue concerns the remaining software resources, such as applications, libraries etc. In this case the CGO, again, defined a rich set of classes, but the defined properties were limited to *installedSoftware* (which notably lacks the definition of domain and range) and *needLib* (which defines a relationship between a *GridApplication* and a *Library*). In our case we needed, on the one hand, a property like *installedSoftware* but with range and domain restricted to the *GridComponent*, and the *Software* classes. On the other hand, we also needed a way to specify the software requirements of a job that could be used to search for a node that the job could be run on.

3.4 CPU requirements

The last of the key issues discovered when analyzing the CoreGRID ontology concerned the way it dealt with the CPU specification. Recent years are characterized by an explosion of processor types. We are dealing not only with vector processors, and multicore processors (which could be treated as extensions of an earlier cache-oriented hardware), but also have to consider potential utilization of hybrid architectures like the Sony/Toshiba CBE (Cell Broadband Engine) or the GPU (Graphical Processing Unit), as well as existence of performance accelerators produced, among others, by the ClearSpeed company. In this context it has been noted that in many cases (especially for more specialized jobs), there may exist very detailed specification of the required CPU configuration. Some jobs may, for example, be optimized for the CBE, while others may be requested to not to be executed on vector processors (e.g. sparse linear algebraic algorithms usually perform badly on such hardware). Similarly, for some CPU and memory intensive applications (e.g. codes for large dense matrices), the size of the L1, L2 or L3 cache (and their arrangement vis-a-vis multiple cores within the processor) might be an important factor in evaluating a resource to execute the task. Apart from that, the number of cores available for the job is becoming an extremely important factor—parallelism available within the application determines how many cores it can take advantage of, and whether having more is going to be beneficial.

Unfortunately, the only property related to the CPU in the CGO was the *clockSpeed*, which notably was defined as a string, which poses an issue in terms of our UI and automated reasoning about preferences. There were no properties related to the CPU architecture, number of cores, or the cache sizes.

3.5 Classes and properties originating from contract negotiations

The second group of issues that needed to be addressed originates from the fact that our ontology has to include classes and properties for the description of terms and parameters to be used in the Service Level Agreement (SLA) negotiations. Obviously, while the above mentioned issues required modifications to the CGO, here we deal with classes and properties that, for obvious reasons, were never considered to be included in the CGO.

As described above, negotiations between agents in the system materialize primarily in two main scenarios: (1) contracting a job to be executed, and (2) joining a team. For the time being we assume that the implementation of these processes will continue to be based on the FIPA Contract Net Protocol (single round [1], or iterated [2]). Therefore, the ontology used in the system needs to contain definitions of required messages and their possible content. Messages that are needed by both processes should include the initial call for proposals, proposal, acceptance or refusal of a proposal, and information about the final result; i.e. the winner of the negotiation process. While assuming utilization of the FIPA-CNP, the ontology should be designed in such a way that it may be able to naturally handle also other one-round and multi-round negotiation scenarios (e.g. the Dutch auction, the sealed first-price auction, etc.). In both SLA negotiation scenarios, negotiation will involve multiple parameters; since in real-life situations there are multiple factors influencing the final decision, e.g. while contracting a job to be executed price is naturally important, however, it may happen that the *User* is willing to pay more money to have the task competed earlier. Therefore, the proposal evaluation will be based on multicriterial analysis. Hence, for each negotiation criteria (parameter) there must exist a property describing it in the ontology. Parameters characteristic for job execution negotiations include, among others, cost, execution start time, penalty for not completing

the job on time, etc. Parameters concerning joining a team include, for instance, revenue, length of contract, possibility of contract extension, and time of availability. During the negotiation process messages with content based on the ontology have to transfer not only the exact parameter values but also constraints to be imposed on specific parameters such as, for instanxce, minimum and maximum length of contract for agent joining a team. Therefore, the ontology should be extended with means of defining constraints such as minimum, maximum value for continuous parameters (e.g. cost of job execution, revenue for a team member), or a possible set of values for discrete parameters (e.g. acceptable CPUs). Moreover, the *User* may want to specify weight of a parameter i.e. not to treat them all as equally important but to create a hierarchy of importance. For instance, *User* looking for a team that will execute her job may decide that the time of execution is the key criterion, while *LAgent* looking for a team to join may decide that guaranteed revenue is more important than frequency of jobs.

4 Proposed solution

In order to overcome the aforementioned issues concerning usability of the *CGO* in its current form, in our project, we had to modify and extend it. Proceeding in this direction, for better modularity, and thus maintainability, we have split the constructed ontology into three (sub)ontologies: (1) AiG Grid Ontology, (2) AiG Conditions Ontology, and (3) AiG Messages Ontology. In the remainder of this section we present these modules in some detail.

4.1 AiG Grid Ontology

The *AiG Grid Ontology* is an extension of the *CGO* that adds additional classes and properties that make specifying the hardware and software configuration of a Grid resource easier. Note that some of these classes and properties have been already defined in our original mini-ontology of the Grid. Classes that it defines—*CPUArchitecture* (and its subclasses), *CPUVendor*, and *Memory* subclasses (*PhysicalMemory*, and *VirtualMemory*)—allow for a more detailed description of hardware capabilities of the node, as well as make it possible to identify compatibility issues between jobs to be executed and resources that are to be used. The proposed extension also specifies a number of properties that join concepts of the CoreGRID ontology. In some cases there is an overlap between the newly introduced properties and these already existing in the *CGO*, but we provided ones that have clearly specified domain and a strongly typed range, which allows us to use them in Java code generation.

In addition to classes, we have introduced some new properties related to them: *hasArchitecture*, *hasVendor*, *hasMemory*. Furthermore, we have added a hierarchy of properties allowing for software configuration specification, these included: *hasInstalledSoftware*, *hasOperatingSystem*, *isRunningOS*; and some additional ones that link the original CGO classes but were missing from the ontology: *hasStorageInterface*, *hasStorageSpace*, *hasFileSystem*.

Finally, the extension includes a number of data properties that can be used to describe classes defined in the *CGO*. These properties include:

- *CPU*—*hasClockSpeed*(in MHz), *hasCores* (number of cores), *hasL1CacheSize*, *hasL2CacheSize*, *hasL3CacheSize*
- *Memory* and *StorageSpace*—*hasAvailableSize*, *hasTotalSize*
- *Software*—*hasVersion*
- *WorkerNode*—*isVirtualized*

4.2 AiG Conditions Ontology

The *AiG Conditions Ontology* contains concepts necessary to describe the conditions of contracts between these entities in our system that are involved in "business" relationships: (1) the *LAgent* representing a *User* offering its resources and the *LMaster* looking for workers, and (2) the *LAgent* looking for a team to execute a job and the *LMaster* offering to run this job. To deal with these two cases, the ontology introduces two new classes: *JobExecutionConditions* and *WorkerContractConditions* and properties describing them. It uses the concepts from the above described *AiG Grid Ontology* and the *Time Ontology* ([4]). Let us now briefly describe classes comprising this ontology and their properties.

4.2.1 JobExecutionConditions

The *JobExecutionConditions* class specifies conditions of the contract between the *LAgent* and the *LMaster* accepting the job to be executed by its team. It has the following properties:

- *jobExecutionTime* – utilizes an instance of the *TemporalEntity* class to describe the period of time during which the job should be executed
- *deadlinePenalty* – the amount of monetary units that the team will pay if it doesn't meet the deadline specified by the *jobExecutionTime* property

- *price* – number of monetary units that the *LAgent* will pay to the team upon completing the task

4.2.2 WorkerContractConditions

The *WorkerContractConditions* class defines the conditions of the contract between the *Worker* and the *LMaster*. It has the following properties:

- *contractPeriod* – the period of time during which the contract holds
- *contractedResource* – the instance of the CGO *GridComponent* class that represents the resource offered by the *LAgent*
- *workerAvailability* – the time (or periods of time) during which the resource will be available for the team; property defined using the *TemporalEntity* class; thus, it is possible to specify, for instance, every Monday or every first week of the month
- *guaranteedUtilization* – the minimum percentage of available time that the *LMaster* guarantees to occupy the *Worker* with paid tasks (not used if payment is based on availability)
- *isContractExtensionPossible* – determines if the parties are willing to extend the contract after the current period ends
- *payment* – the amount of monetary units the *LMaster* agrees to pay the *LAgent* for each hour of resource utilization (or for each hour of availability)

4.3 AiG Messages Ontology

The *AiG Messages Ontology* is utilized internally by the proposed solution and contains definitions of messages that can be exchanged between agents. Content of messages is based on terminology defined in the aforementioned ontologies. At the moment we are in the process of evaluating different approaches to ontological description of parameter constraints such as these described in section 3.5. Therefore, in the following list of classes from the *AiG Messages Ontology* we will omit concepts related to the constraint definition.

The *AiG Messages Ontology* is split into two major classes: 1) *JobExecutionMessage*, and 2) *TeamJoiningMessage*.

4.3.1 JobExecutionMessage

Subclasses of the *JobExecutionMessage* describe the structure of messages related to the process of negotiations the terms of job execution, by the resources of the team. Following is a brief description of these concepts:

- *JobExecutionEnquiry* – a message sent from the *LAgent* to the *LMaster* as a *Call For Proposals*; it contains the requirements related to the hardware and software configuration of a desired resource (constraints on properties from the *AiG Grid Ontology*), as well as constraints on the contract conditions (based on the *JobExecutionConditions* class)
- *JobExecutionOffer* – a message sent from the *LMaster* to the *LAgent* containing the proposal from the team; it is described by an instance of the *JobExecutionConditions* class through the *proposedJobExecutionContract* property
- *JobExecutionRefusal* – a message informing the *LAgent* that the team will not accept the job
- *JobExecutionOfferAccept* – a message informing the *LMaster* that its team has been selected to execute the job
- *JobExecutionOfferReject* – a message informing the *LMaster* that it's offer has been rejected by the *LAgent*
- *JobExecutionCounterOffer* - a message sent from the *LAgent* containing modified details of the contract that we received in *JobExecutionOffer* (used in case of multi-round negotiations).

4.3.2 TeamJoiningMessage

The *TeamJoiningMessage* is a superclass of all messages related to the process of an *LAgent* applying for a position in the *LMaster*'s team. Its hierarchy contains of the following classes:

- *TeamEnquiry* – a message sent from the *LAgent* to the *LMaster* as a question about the terms on which it might be accepted; it utilizes the *offeredResource* property and contains an instance of the *GridComponent* class which describes the configuration of the resource it is offering; it will also contain a definition of constraints on properties from the *WorkerContractConditions* class that specify what the *LAgent* would like to include in the contract
- *TeamOffer* – a message sent from the *LMaster* containing details of the contract it can propose to the *LAgent*; it contains an instance of the *WorkerContractConditions* with specific values of contract parameters
- *TeamRefusal* – a message sent by the *LMaster* to the *LAgent* in case the team does not want that *LAgent* to join the team

- *TeamOfferAccept* – a message confirming that the *LAgent* has accepted the contract and decided to join the team
- *TeamOfferReject* – a message rejecting the offer made by the *LMaster*
- *TeamCounterOffer* - a message sent from the *LAgent* containing modified details of the contract that we received in *TeamOffer* (used in case of multi-round negotiations)

5 Concluding remarks

The aim of this note was two-fold. First, we have discussed the reasons why the CoreGRID ontology has to be modified and extended to become the centerpiece of our project aiming at developing an agent-based intelligent high-level Grid middleware. Second, we have outlined the way that this goal has been achieved. Specifically, we have presented in some detail how the three resulting (sub)ontologies have been created. The first of them, the *AiG Grid Ontology*, is a direct modification and extension of the CGO, while the remaining two (the *AiG Conditions Ontology* and the *AiG Messages Ontology*) result from the need to facilitate autonomous agent-based contract negotiations. Currently, we are in the final stages of ontology design; considering technical details of dealing with constraints. As soon as this process is completed the proposed ontology will be made available at http://sourceforge.net/projects/gridagents/. At this stage it will be placed within our system, replacing our current mini-ontology.

Acknowledgments

Work of M. Ganzha and M. 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). Collaboration of the Polish and Bulgarian teams is partially supported by the *Parallel and Distributed Computing Practices* grant. Collaboration of Polish and French teams is partially supported by the PICS grant *New Methods for Balancing Loads and Scheduling Jobs in the Grid and Dedicated Systems.*

References

[1] FIPA Contract Net Protocol Specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`.

[2] FIPA Iterated Contract Net Interaction Protocol Specification. `www.fipa.org/specs/fipa00030/PC00030D.pdf`.

[3] Jena—A Semantic Framework for Java. `http://jena.sourceforge.net`.

[4] OWL-ontology. `http://www.w3.org/2006/time`.

[5] SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query`.

[6] M. Dominiak, M. Ganzha, M. Gawinecki, W. Kuranowski, M. Paprzycki, S. Margenov, and I. Lirkov. Utilizing agent teams in grid resource brokering. *International Transactions on Systems Science and Applications*, 3(4):296–306, 2008.

[7] M. Drozdowicz, M. Ganzha, M. Paprzycki, R. Olejnik, I. Lirkov, P. Telegin, and M. Senobari. Ontologies, agents and the grid: An overview. In B. Topping and P. Iványi, editors, *Parallel, Distributed and Grid Computing for Engineering*, pages 117–140. Saxe-Coburg Publications, Stirlingshire, UK, 2009.

[8] I. Foster, N. R. Jennings, and C. Kesselman. Brain meets brawn: Why grid and agents need each other. *Autonomous Agents and Multiagent Systems, International Joint Conference on*, 1:8–15, 2004.

[9] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2):30–37.

[10] W. Kuranowski, M. Ganzha, M. Gawinecki, M. Paprzycki, I. Lirkov, and 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.

[11] W. Kuranowski, M. Paprzycki, M. Ganzha, M. Gawinecki, I. Lirkov, and S. Margenov. Agents as resource brokers in grids—forming agent teams. volume 4818 of *LNCS*, pages 472–480, Berlin, 2007. Springer.

[12] H. Nwana and D. Ndumu. A perspective on software agents research. *The Knowledge Engineering Review*, 14(2):1–18, 1999.

[13] W. Xing, M. D. Dikaiakos, R. Sakellariou, S. Orlando, and D. Laforenza. Design and development of a core grid ontology. In *Proc. of the CoreGRID Workshop* Integrated research in Grid Computing, pages 21–31, November 2005.